Introducing JavaScript Frameworks

Introducing Shaun & JavaScript Frameworks

Introducing Me

- Education
- Experience
- Personality
- How to communicate with me
    - Slack invite
    - Email

Introducing JavaScript Frameworks

We will be utilizing JavaScript and its many frameworks to create a complete application solution from server to client in a single page format.

- Frameworks are
    - Existing code structures that make building applications simpler
        - Bootstrap
        - HTML5 Boilerplate
        - MVC Frameworks
            - PHP's Laravel, CakePHP, Symphony, SparkyCMS
            - Ruby's RoR and Sinatra
            - JavaScript's Express Generator, Meteor, ReactJS, Angular, Ember, Vue, etc...
    - Create a best-practice code structure and attempt to enforce conventions
- The JavaScript frameworks we'll be using
    - NodeJS - a server-side scripting solution
    - ExpressJS - a server-side routing solution
    - Mongoose - a JavaScript object data modeller (ODM) that will allow us to interact easily with MongoDB
    - ReactJS - a client-side library that provides a solution to single-page application development
- Some other libraries we'll explore
    - Twitter's Bootstrap
    - TinyMCE WYSIWYG editor
    - React Router
    - Bcrypt Authentication (or Passport. I haven't decided yet)
    - Axios HTTP Client for working with ReSTful API requests

Our Example Project

Our example project will be a blogging platform that allows for multiple blog categories, blogs, authors, and anonymous commenting. We will (hopefully) end with a WYSIWYG editor to make adding new blogs interactive and simple. Ideally we will support a feature image as well.

Overview of the Next 14 Weeks

1. Intros and Reviews
2. Simple Node Server and Deploying to Heroku
3. Understanding MVC (working with routing, views, and controllers)
4. MongoDB and Models Part 1
5. MongoDB and Models Part 2
6. Basic Authentication Strategy
7. Midterm
8. Introducing ReactJS
9. Keeping it DRY in React
10. Working with React and NodeJS using APIs Part 1
11. Working with React and NodeJS using APIs Part 2
12. Authentication in React
13. Utilizing TinyMCE to add more to our app
14. Final

The Gradeables

NOTE: This is subject to change based on class needs and conduct

Qty	Item	Description	Grade
10	In-class Quizzes	The in class quizzes will be presented at the beginning of each class allowing students to practice the lesson from the prior class and demonstrate their understanding. Missed quizzes **cannot** be taken. (1% each)	10%
5	Labs	Each lab will demonstrate a complete concept: • Lab 01 - Constructing a simple node server and deploying it • Lab 02 - Basic create/read operations using Mongoose • Lab 03 - Basic update/delete operations using Mongoose • Lab 04 - Simple ReactJS setup and deployment • Lab 05 - Working with API data in ReactJS Labs can be completed in groups. All labs will be deployed to Heroku (or wherever you choose) and will have a GitHub counterpart.	20%
1	Project	The project will be the bulk of your grade and it will be broken into 2 parts: 1. The NodeJS half will be due before the midterm and represent the first 6 weeks of the course 2. The ReactJS half will integrate seamlessly with the NodeJS half and represent the last 6 weeks of the course Projects will be group based (with an option to do it individually). Groups will be a max size of 4 people. The project will be deployed to an online host and will have a GitHub counterpart	40%
2	Major Tests	There will be 2 major tests: 1. The Midterm will encompass our NodeJS component and be valued at 15% 2. The Final will encompass our ReactJS component and be valued at 15% The tests will contain 2 components: a question portion and a practical portion. You will have 1 hour to complete the question portion and 1 hour 50 minutes to complete the practical portion. The question portion will be **individual** and the practical portion will be **group based** if you choose	30%

Questions?

Expecations

Class Etiquette

- Be respectful to me and other students (don't talk when I'm talking)
- Be on time (we start at 2 minutes pass the hour)
- Participate in class discussions
    - Helps me learn who you are
    - Helps to identify yourself as a competent colleague
    - Promotes class engagement
    - Makes the 3 hours fly by and not drag on into a horrible soul-sucking abyss of nothingness
- Complete the in-class examples
    - Helps to reinforce the learning outcomes
    - Provides immediate feedback on your understanding

Group Work Requirements

- **Sign up by entering your group information in the provided Google Sheet under Blackboard -> Resources -> Group Member Sign-Up**
- Group-based assignments require a group assessment from each member. The group assessment will be done through a rating application that will be provided later
- Group members are required to be equal contributors and therefore a doc must be submitted with any group-based assignment detailing what each member did
- **You are not required to work in a group but it is recommended**
- Group dynamics (disagreements, conflicts, etc...) can occur and are to be dealt with in this format:
    1. Attempt to resolve internally by declaring a mediator and presenting both arguments constructively
    2. Vote for a solution
    3. Involve Shaun only if the above does not work
- Group members found not communicating/working can be ejected from a group **only** after a discussion with Shaun

---

Misconduct

Due to the overwhelming amount of misconduct that has been occurring recently, my misconduct rules have changed and become much stricter.

How to identify if you've committed misconduct:

- You copied a solution, in whole or in part, from the internet and submitted it as your own work
- You copied a solution, in whole or in part, from a colleague and submitted it as your own work
- You provided a colleague a solution, in whole or in part, and they submitted it as their own work
- You communicated to colleagues during a quiz/test or to colleagues who have not taken the test you have taken about the test content

What will happen if you commit misconduct:

- You fill out a misconduct form with me
- The Registrars Office will be contacted to see if you've committed misconduct before
- **IF YOU HAVE NOT**
    - You will receive a zero on your assignment/lab/quiz/test
    - The misconduct will be filed with the Registrars Office
    - I will be marking future assignments/labs/quizzes/tests with considerably more scrutiny
    - Past assignments/labs/quizzes/tests will be reopened and remarked with more scrutiny
- **IF YOU HAVE**
    - You will receive zero in the course
    - The misconduct will be filed with the Registrars Office
    - If you have multiple misconducts you will be expelled from the program and will have to retake the program in its entirety in the future

Why do I care if commit misconduct:

- Our industry is cheapened by students who have diplomas and can't complete the work
- It reflects upon Georgian College as a credible school by releasing incapable students
- It shows you're a student who doesn't deserve the position you've received when there is a massive waiting list of potential students to take this program

Projected Class Structure

Our Class Format

1. (5 min) - Previous class review
2. (10 min) - Quiz on previous class learning outcome
3. (10 min) - Introduction to today's class learning outcome
4. (20 min) - Simple activity reflecting the learning objective
5. (10 min) - Break
6. (60 min - 120 min) - Elaborate activity adding a new component to our example project
7. (20 min) - Discussion/question period to help fully understand the learning objective
8. (10 min) - Break (will obviously fall in the middle of the elaborate activity session)

Our Example Project - In-Depth Look

Our Project - A Week-by-Week

Our example project will be a blogging platform. Below is the breakout by week of what we'll attempt to accomplish:

1. Just some basic review
2. Setting up our Node server and deploying to Heroku
3. Creating some basic routes, views and controllers
    - A home page
    - An about page
    - A contact page
4. Creating the logic for adding a new author and draft blog post
5. Creating the logic for updating a draft blog post and deleting a post
6. Adding user authentication to our project
7. Converting our existing pages over to ReactJS Part 1
8. Converting our existing pages over to ReactJS Part 2

9. Converting our NodeJS app to API only and getting our ReactJS side to communicate with it
10. Creating a commenting and rating system
11. Connecting our authentication using JavaScript Web Tokens (JWT)
12. Converting our blogging block to TinyMCE (or some kind of WYSIWYG editor)

Styling/Presentation

To simplify styling we will use some available frameworks:

- We will be using Twitter's Bootstrap for most of the styling
- We will be using Font-Awesome 5 for our icons
- We will use ListJS for any sorting/filtering
- We will use Select2 for any cool select boxes (tagging)
- We will use DatePicker for our scheduled publish dates

BREAK

Introducing NodeJS

Introducing NodeJS

- When Google announced Chrome and its new **V8 JavaScript engine** in late 2008, it was obvious that JavaScript could run faster than before—a lot faster.
- V8's greatest advantage over other JavaScript engines was the compiling of JavaScript code to native machine code before executing it.
- This and other optimizations made JavaScript a viable programming language capable of executing complex tasks.
- Developers decided to try a new idea: non-blocking sockets in JavaScript.
- They took the V8 engine, wrapped it with the already solid C code, and created the first version of **Node.js.**
- **Node.js** uses the event-driven nature of JavaScript to support non-blocking operations in the platform, a feature that enables its excellent efficiency.
- JavaScript is an **event-driven language**, which means that you register code to specific events, and that code will be executed once the event is emitted.
- This concept allows you to seamlessly execute asynchronous code without blocking the rest of the program from running.
- Let's examine how this works with some help from our friends at Code School: https://www.youtube.com/watch?v=GJmFG4ffJZU

Installing NodeJS

- browse to nodejs.org, install right on home page & auto-detects OS
- download & install
- no real gui so we actually use the command line and then a simple text editor
- installer also installs Node Package Manager (npm). similar to WP plugin library or .net NuGet, npm allows us to easily install prebuilt code libraries
- some popular npm packages we will use:
  - express - primary http library
  - mongodb / mongoose - accessing a MongoDB object db (NOT a relational db, separate package for that)
  - on subject of db's, node will work w/rel db's but generally it's not that well suited to it. We will look at nosql db's instead which play very well w/node
  - jade (html templating)
- unofficial style guide available here: https://github.com/felixge/node-style-guide (also posted to links folder on BB)

Testing Our Installation

- Open an IDE
- Create a new file
- Add the line

  console.log("Hello World");

- Save the file **helloworld.js**
- Open command line and navigate to the folder that contains **helloworld.js**
- Run **node helloworld.js**
- The console should output **"Hello World"**

JavaScript Review

[A re-introduction to JavaScript (JS tutorial)]

Variables

Variables declared with **var** are scoped to the nearest function block.

```
// var
function varring () {
  if (true) {
    var youCanSeeMe = 'Bob';
  }

  console.log(youCanSeeMe);
}
varring();
```

Variables declared with **let** are scoped to the nearest block. This is very powerful, but can be a gotchya when you're attempting to use a variable defined in a block elsewhere.

```
// let
function letting () {
  if (true) {
    let youCanSeeMe = 'Bob';
  }

  console.log(youCanSeeMe);
}
letting();
```

Both variables can be defined globally outside a function block. However, it is important to note, that the variables defined with **let** will not be available as properties on the **window** object.

const variables are constant. This means they cannot be redeclared or reinitialized.
They are also block-scoped, like **let**.

```
function consting () {
  if (true) {
    const youCanSeeMe = 'Bob';
  }

  console.log(youCanSeeMe);
}
consting();
```

Arrays

Arrays are available in every language, but in Javascript they're not type restricted, and all arrays are dynamically sized.

```
// arrays
let dc = ['Batman', 'Wonder Woman', 'Superman', 'Flash', 'Cyborg', 'Green Lantern'];
let marvel = ['Captain America', 'Black Widow', 'Incredible Hulk', 'Hawkeye', 'Thor', 'Iron Man',
'Vision'];

// current length
console.log( dc.length );
console.log( marvel.length );

// adding new elements
dc.push( 'Darkseid' );
marvel.push( 'Thanos' );

// new length
console.log( dc.length );
console.log( marvel.length );
```

Objects

You may hear people say that "Everything is an object in Javascript". While not **everything**in Javascript is an object, almost everything is. Variables, functions, and arrays are all objects. This provides a lot of flexibility to how we work with our program.

```
let student = {
  id: 200230778,
  name: 'Shaun',
  age: 39
};
student.output();
```

In the above example we've created an object with some information in it. We aren't limited to just data though. We can add functions, arrays, prototypes, whatever we really want within the object. We'll look at this again further on.

Functions

Functions are essential to DRYing up code. In Javascript we can define functions directly or anonymously.

```
// direct function declaration
function createAStudent ( id, name, age ) {
  return {
    id: id,
    name: name,
    age: age
  }
}
console.log( createAStudent( 12345, 'Shaun', 39 ) );
```

```
let student = function ( id, name, age ) {
  return {
    id: id,
    name: name,
    age: age
  }
}
console.log( student( 12345, 'Shaun', 39 ) );
```

Control Statements

We have all experienced control statements by this point. Basically they control the flow of our application. These can be conditional statements, functional statements, and iterative statements.

Decision/Conditional statements are how we control the direction we want a program to go. They allow us to evaluate an expression and determine based on a boolean result what to do next:

```
// standard if/else condition
let flag = true;
if ( flag === true ) {
  console.log( 'do this...' );
} else {
  console.log( 'do this instead...' );
}

// single line
if ( flag === true ) console.log( 'do this...' );

// implied unless false
if ( flag ) console.log( 'do this...' );

// ternary
console.log( ( flag ) ? 'do this...' : 'do this instead...' );

// logical operators
let doThis = null;
let doThisInstead = 'ohhh yeah';
console.log( doThis || doThisInstead );
doThis = true;
console.log( doThis && doThisInstead );

// switch statements
let name = 'Bob';
switch ( name ) {
  case 'Bob':
    console.log( 'Hi Bob' );
    break;
  default:
    console.log( 'Hi' );
    break;
}

// switch statements made cool
switch( true ) {
  case (5 > 3):
    console.log( 'Yes it is' );
  case (10 > 7):
  case (6 < 10):
    console.log( 'Hells yeah' );
  default:
    console.log( 'Guess we need some break statements' );
}
```

Repetition structures in a programming are essential when we're trying to read files, process arrays, listen for events, basically anything that involves the need to repeat ourselves. Javascript gives us several ways to iterate:

```
// Repetition Control Statements
// the for loop
for ( let i = 0; i < 10; i++ ) {
  console.log( 'Yay for ' + i );
}

// short hand, but you are responsible for
// handling the terminating condition
let i = 0;
let tCon = true;
for ( ; tCon; ) {
  console.log( 'Yay for ' + i );
  if ( i == 10 ) {
    tCon = false;
  }

  i++
}

// the while loop
let count = 0;
while ( count < 10 ) {
  console.log( 'Yay for ' + count );
  count++;
}

// the do while loop
let doCount = 0;
do {
  console.log( 'Yay for ' + count );
  count++;
} while ( count < 10 );

// iterating through arrays
let days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'];
// the basic way
for ( i = 0; i < days.length; i++ ) {
  console.log( days[i] );
}

// the cleaner way
for ( let day of days ) {
  console.log( day );
}

// iterating through objects
let months = {
  Jan: 'January',
  Feb: 'February',
  Mar: 'March',
  Apr: 'April',
  May: 'May'
}

for ( let key in months ) {
  // we should be sure the property is of this object
  if ( !months.hasOwnProperty( key ) ) continue;

  console.log( key, months[key] );
}
```

BREAK

Blocking VS Non-Blocking Code

Callbacks

Synchronous Callbacks

We have seen in some examples of setting up the Node server, that we have the ability to pass functions as arguments to other functions. These are generally referred to as callbacks (but you may also see them called higher order functions).

The idea of a callback is for it to be executed at a specific time. The convenience of a callback is it has current functional scope and state. We generally execute callbacks once our application has identified that an event has occurred and it now wants to do something in response.

It is important to understand that while callbacks are used a lot in asynchronous programming, they are not exclusive to it. There are many reasons that you may want to maintain context and state as you execute your application, and callbacks allow for that.

Let's look at an extremely basic example of a callback function:

```
function myTask ( my_callback_func ) {
  console.log( "I have completed whatever it was that I wanted to do." );
  // I am now executing my callback function
  my_callback_func();
}

myTask( function () {
  console.log( "My turn to work!" );
});
```

As you can see, our function logic executed the callback by simply calling the symbol as a function. If, for any reason, we hadn't passed a function, this would result in an error:

```
▶Uncaught TypeError: my_callback_func  VM9327:5
is not a function
    at myTask (<anonymous>:5:3)
    at <anonymous>:1:1
```

This callback was synchronous. This means that our code executed in a blocking manner, not allowing other things to occur while it was processing. This is not the general norm for using a callback. Usually we use callbacks when an asynchronous action has occurred and we are ready to work with its response.

Asynchronocity

Before we get into **Asynchronous Callbacks** it is important that we understand the difference between synchronous and asynchronous code. In order to do this, we can use a handy Javascript function known as **setTimeout()** that allows us to delay when a piece of code will fire:

```
function output( message ) {
  console.log( message );
}

// setTimeout( function name, delay in ms, parameters );
setTimeout( output, 5000, 'I want to go first!' );
output( 'No, I want to go first!' );
```

"No, I want to go first!" executed before the "I want to go first!" statement due to that Javascript is **asynchronous**. This means it won't wait. It won't **STOP THE WORLD** as our friend from the video stated.

This is hands down the concept that confuses most new Javascript programmers. Most languages (unless you learn threading or forking) are synchronous. This means calls that take time, do exactly that: **they take time**. They stop our application from continuing and we have to wait before it will continue on. This is the standard way most programmers learn to program. However, once you begin working with Javascript, you start to realize the frustration of working with synchronous code, and you begin to appreciate the asynchronous nature of Javascript.

Callback Hell

Callback Hell is the result of needing several things to occur synchronously that want to occur asynchronously. The result is a slew of nested callbacks:

```
// callback hell
var https = require ( 'https' );

let getInsult = function ( callback ) {
  let body = '';

  // 1
  let req = https.get( 'https://insult.mattbas.org/api/insult',  function ( res ) {
    // 2
    res.on( 'data', function ( chunk ) {
      body += chunk;
      // 3
      callback( body );
    });
  });
}

getInsult( function ( body ) {
  console.log( body );
});
```

3 callbacks to achieve what we want. And it's very difficult to follow what's happening for people reading your code. Let's walk through this:

1. The first callback is the one that is fired when https.get has completed its request to the url we passed. It calls our passed function passing in the response object as an argument
2. The second one is called once we receive the 'data' packet. We call function and pass the data chunk as an argument to the callback function
3. The third callback is made so we can pass our parsed data to the callback. This allows the developer who's calling the function to trigger a response once they have the body

Callback hell is the reason **promises** now exist. It is cleaner and much easier to read when writing code.

Simple Node Server and Deploying to Heroku

Blocking VS Non-Blocking Code

What is Blocking and Non-Blocking Code?

Blocking (**synchronous**) code is code that prevents execution (stops the world) of other code till it has completed its operation. **Most** operations are blocking in JavaScript, but there are times where operations are **non-blocking**.

Non-blocking (**asynchronous**) code is code that **does not** prevent the execution of other code. It will break it's operations into events and add them to the stack as needed. This allows other events to complete their executions while the asynchronous code catches up. The following are examples of when asynchronocity occurs:

Asynchronous Operations

- HTTP requests (the response is handled asynchronously to avoid waiting for operations to be completed on the foreign server)
- File requests (these can be blocking, but usually file handling is asynchronous due to the unknown size or potential needs of the file)
- setInterval and setTimeout (these are asynchronous so they can perform logic repeatedly or after a delay. If they were synchronous you would be stuck in a loop or a queue till they completed)
- Event registered functions (any function that is bound to an event like **click**, **mousedown**, **focus**, etc are asynchronous so they can be handled once the event has occurred)

Let's look at an example of blocking and non-block code:

```
// This function takes one argument and console logs a statement
function sayHi(name) {
  console.log(`Hello ${name}!`);
}

// The below code is blocking. Each line will happen synchronously
sayHi(`Keval`);
sayHi(`Devon`);
sayHi(`Connor`);

// The below code is asynchronous. The lines sayHi(`Darryl`) and sayHi(`Ilia`)
// will execute before the setInterval lines
setInterval(sayHi, 2500, `Michael`);
setInterval(sayHi, 5000, `Gagandeep`);
sayHi(`Darryl`);
sayHi(`Ilia`);
```

Callbacks

Callbacks are essential to asynchronous programming but not exclusive to it. Simply put, ***a callback is a function that is passed as an argument to another function to be called at a later time***. Callbacks are probably one of the more confusing programming paradigms for beginning programmers, but really they're quite easy to use.

Let's look at an extremely basic example of a callback function:

```
// Our callback function that we'll pass as an argument later to our greeting function
function farewell(name) {
  console.log(`Goodbye ${name}.`);
}

// Creating parameters for our name and callback function
function greeting(name, callback) {
  console.log(`Hello ${name}.`);

  callback(name);
}

// Calling our greeting function, passing in our name and our callback arguments
greeting(`Shaun McKinnon`, farewell);
```

The above example illustrates a very basic example of using a callback. The benefits of a callback are as follows:

- Code separation and modularity (the **greeting** function doesn't need to care about what the **callback** function does)
- Priority execution (the **callback** can be executed where needed which is more beneficial to asynchronous operations)
- Provides an order-of-operation structure where one function can invoke another of it has executed

Using Callbacks for Asynchronous Operations

Callbacks are often used to combat asynchronocity. They allow us to execute code at the correct moment in time where we need to. For example, because an HTTP request can take an unknown amount of time, it is optimal to that HTTP requests are asynchronous. The issue lies in not knowing when to return our response. Let's take a look at the example below:

```
// https is a module that is apart of the core Node API
const https = require('https');

// Getting an insult from the infamous Matt Bas Insult API
const getInsult = function (callback) {
  const url = `https://insult.mattbas.org/api/insult`;

  https.get(url, function (client) { // callback 1
    client.on(`data`, function (data) { // callback 2
      callback(data.toString()); // callback 3
    });
  });
}

// We call getInsult and pass an anonymous function as a callback
// so we can utilize the returned data object
getInsult(function (insult) { // callback 4
  console.log(insult);
});
```

The above is also a good example of **callback hell** a term given to code that uses a large amount of callbacks. Lot's of callbacks render the code difficult to read, as you can see. Let's look at what each callback is doing:

1. The first callback is used to return the client agent to our function body from the https request
2. The second callback is used to capture the data once the **data** event has been executed by the client
3. The third callback is used to pass our data to a new function scope
4. The fourth callback is used as an argument to our getInsult function. This allows us to capture and use the returned **data**

Promises

Promises

Promises are meant to help callbacks be more readable. By encapsulating our function logic in a promise, we can then provide 2 new methods helping our function to make more sense: **.then** and **.catch**.

.then will execute regardless of what our response is from our promise. If we **resolve** our promise (meaning our function was successful), **.then** will be called. If we **reject** our promise, then **.then** will also be called. If want to instead funnel our rejection pass the **.then** we can either not reject it, or we can use **.catch** to aid in handling the error.

You may be asking where do **reject** and **resolve** come from? Both of these are callback functions that get passed to a **new Promise**. Let's convert our last example into a new a promise wrapped function.

In our IDE, we'll use the following code to recreate our last example using promises:

```
// https is a module that is apart of the core Node API
const https = require('https');

// Getting an insult from the infamous Matt Bas Insult API
const getInsult = function (callback) {
  const url = `https://insults.mattbas.org/api/insult`;

  // a new promise takes a callback as an argument that
  // has 2 callbacks as parameters: resolve and reject
  return new Promise(function (resolve, reject) {
    // We make our request to Matt Bas' url
    const req = https.get(url, function (client) {
      // When we have the data, we resolve
      client.on(`data`, function (data) {
        // Think of this as our successful callback
        resolve(data.toString());
      });

    });

    // If the request breaks, we can catch the error
    req.on(`error`, function (error) {
      // Think of this as our fail callback
      reject(error);
    });
  });
}

// Calling getInsult using our new methods
// .then will catch everything, but because we have
// .catch, we can ensure errors go to .catch and our
// successes go to .then
getInsult()
  .then(function (data) {
    console.log(`Resp:`, data);
  })
  .catch(function (error) {
    console.log(`Error:`, error);
  });
```

Promises can also be chained together allowing for them to execute in a sequential format. Promises are fantastic, but **Async/Await** makes life even better.

QUIZ

The quiz is 5 questions and you have 2 attempts. **ATTEMPT THIS ON YOUR OWN AS IT SHOWS YOUR UNDERSTANDING!**

BREAK

Introducing ES6 Syntax

ES6 Cheat Sheet

What is ES6?

ES6 stands for ECMA Script 6. ECMA is an organization that standardizes information. Because Javascript is implemented differently in various browsers, there has to be a standard and specification that they must follow to ensure the developers have a

common environment to work with. However, updating browsers to support every new standard can be costly, and therefore browsers will adopt when they choose. They also aren't required to update old browsers with the new standard, so it is very possible that new code will not work properly in older browsers.

Transpilers to the Rescue!

Transpilers allows us to use newer code functionality, regardless of support for that functionality, by transpiling the new code into legacy compatible code. This means that older browsers, and browsers that are taking their time, will still work.

However, we now have an extra step in our process. We must transpile in order to ensure we are cross browser compliant. That isn't a big deal considering we can add transpilers to our build process, which will result in us always having cross-browser compliant code, regardless of what we code.

Handy ES6 Stuff

We already showed how we can use **let** and **const** instead of **var**. However, ES6 introduces many other really cool features that make writing Javascript simple:

```
// no longer need IIFEs (Immediately Invoked Function Expressions) thanks to block scope :)
(function () {
  var myScopedVar = "Bob";
}());

// now becomes
{
  let myScopedVar = "Bob";
}
```

IIFEs are commonly used to maintain block scope. This aids in avoiding namespace collisions.

Arrow Functions are very handy, and not just because they're shorter to write than full functions. They maintain also maintain the scope of the keyword **this**. Many languages use the keyword **this** to represent the current scoped object. It gives you access to defined properties and methods and is fundamental to object oriented programming. However, in Javascript **this** changes scope to reference its current function container. That means if you call **this** from within a callback it is referring to its callback and not the surrounding function.

Many programmers have overcome this issue by assigning **this** to **that** or **self** or some other arbitrary variable that will maintain its scope for them. This also isn't ideal as you want **this** to be contextual. **Arrow Functions** do not have a sense of **this** in their own context. They pass **this**through with containing context. We definitely need an example:

```
function Person() {
  var that = this;
  that.age = 0;

  setInterval( function growUp() {
    // The callback refers to the `that` variable of which
    // the value is the expected object.
    that.age++;
    console.log( that.age );
  }, 1000 );
}

let peep = new Person();
```

The above may seem odd but is essentially a class with a constructor. The constructor is setting the property of **that** equal to **this** so it will maintain context when given to the callback function **growUp**, otherwise **growUp** will throw a reference error when we attempt to access **this.age** as it isn't present within its context.

Below we attempt the same example, but instead use the **arrow function**:

```
function Person(){
  this.age = 0;

  setInterval(() => {
    this.age++; // |this| properly refers to the person object
    console.log( this.age );
  }, 1000);
}

let peep = new Person();
```

In the above example, we are passing an **arrow function** as the callback. The **arrow function** maintains the context of **this** throughout the body of the function. Much cleaner. Much easier to read.

Arrow functions can be a little confusing because you will see them in different syntaxes. Below is a table showing the various ways you'll see them formatted and what they mean:

Function Syntax	What's it's Doing
`(param1, param2) => { return param1 + param2; }`	The () holds the parameters. The arrow then passes those parameters into our function block, where we are returning them concatenated.
`param1 => { return param1; }`	If you have only 1 parameter, you can forgo the parenthesis.
`param1 => param1;`	If you only have one line to execute you can do it inline. It is implied you want to return it though. The example will pass param1 in as an argument to the function block and then the function block will return param1.
`() => { return "hi"; }`	If you have no parameters, you must use a set of parenthesis to signify it's an arrow function.
`() => "hi";`	This will return the value "hi".
`(() => { return "hi"; })();`	This is an instantly invoked function execution, or IIFE (iffy) for short. This function will automatically run and not be executed. It is most commonly seen when either encapsulating scope or creating an async operation.

Mapis very cool method of array. It allows us to walk through the array and perform operations on each element:

```
['me', 'you', 'them'].map(n => console.log(n.length))
```

The above is an inline expression. Below is a full operation:

```
['me', 'you', 'them'].map(n => {
  console.log( n.length )
})
```

Template Literals (string interpolation) are in so many languages. Some even do it right (I'm lookin' at you Ruby). Template Literals allow you to embed expressions in strings. These expressions will be parsed within the string, making it very easy to embed variables, calculations, function calls, method calls, properties, etc...

```
let name = "Shaun McKinnon"
let age = 39

let text = `${name} is ${age - 20} years old.`
console.log( text );
```

Destructuring is one my most favourite constructs available in ES6. It makes working with arguments more readable. The goal is to get to a point where commenting isn't necessary because the syntax is understandable. **Destructuring**allows us to extract values and then immediately store them in variables that have a more meaningful syntax:

```
let [day, month, year] = [21, 5, 2018]
console.log(day, month, year)
```

The above **destructures** the values in the array. We can now access those values with the more contextual **day**, **month**, and **year** variables. We can literally call the variables whatever we want (following variable naming rules).

```
let Student = {
  name: 'Shaun',
  id: 456321,
  age: 39
}

function myFunc ( {id, name, age} = options ) {
  console.log( name, id, age )
}

myFunc( Student )
```

In the above code, we **destructure** the options and pull out the name, id, and age from the passed object. Keep in mind, we MUST pass in an object that has those values, or we will wind up with variables being undefined.

Also notice that the order of the destructuring isn't the same as the order of the keys in the object. This is important because when destructuring an object is uses the keys to match up the variables. That means we don't have the flexibility in naming the variables like we did in the array. They MUST be the same as the key names.

You can read more about desctructuring. It is a very power feature that helps make code a lot more human readable.

Classes

Classes is so important it gets its own heading. Prior to ES6, we had to this fun:

```
// This is basically our class and constructor
function Student ( id, name, age ) {
  // these are the properties
  this.id = id;
  this.name = name;
  this.age = age;
}

// This is a method of the class
Student.prototype.output = function () {
  console.log( this.id, this.name, this.age );
};

// here's instantiation of the class
( new Student( 1234, 'Shaun', 39 ) ).output();
```

Clear as the windows in a sketchy bar, right? This is prototyping. A very different approach to building 'classes' in Javascript.
If you're looking for a more traditional version of classes, look no further than our friend, ES6:

```
class Student {
  constructor ( id, name, age ) {
    this.id = id
    this.name = name
    this.age = age
  }

  output () {
    console.log( this.id, this.name, this.age )
  }
}

( new Student( 1234, 'Shaun', 39 ) ).output()
```

Annnnnnd, if you want inheritance, you can use the keyword **extends**:

```
class StudentGroup extends Student {}
```

ES6 and ES7 and ES8 are making huge changes to how we code Javascript. These massive changes are going to make a more approachable language for the layman and will definitely increase its popularity.

Lastly (this is a bonus) I have shown you the ES6 for/of and for/in loops. The for/of loop iterates over the elements in an array. The for/in loop iterates over the keys in an object. that's often not that usable, as you are more likely going to utilize the values, not the keys. Below is an example of how you can output the values:

```
for (let [key, value] of Object.entries(obj)) {
  console.log(`${key}: ${value}`);
}
```

In the example we use **Object.entries** which converts our object to a nested array of key/values. We then use **destructuring** to cast each value into a key and a value variable.

BREAK

Introducing NPM (Node Package Manager)

What is a Package Manager?

- Package Managers provide a simplified way to add, update, and remove modules (vendor libraries) to your environment
- Apt, WGET, Brew, Yum, Yarn, RubyGems, Composer, and NPM are examples of package managers
- NPM is the preferred package manager for NodeJS and comes with it when you install Node
- **NPM** has the following main features:
  - A registry of packages to browse, download, and install third-party modules https://www.npmjs.com/
    - You can search for packages that may help your application
    - Packages are solutions that can help mitigate reinventing the wheel
  - A CLI tool to manage local and global packages
    - This is accessible through your terminal or command line tool in your OS

So how do I get NPM?

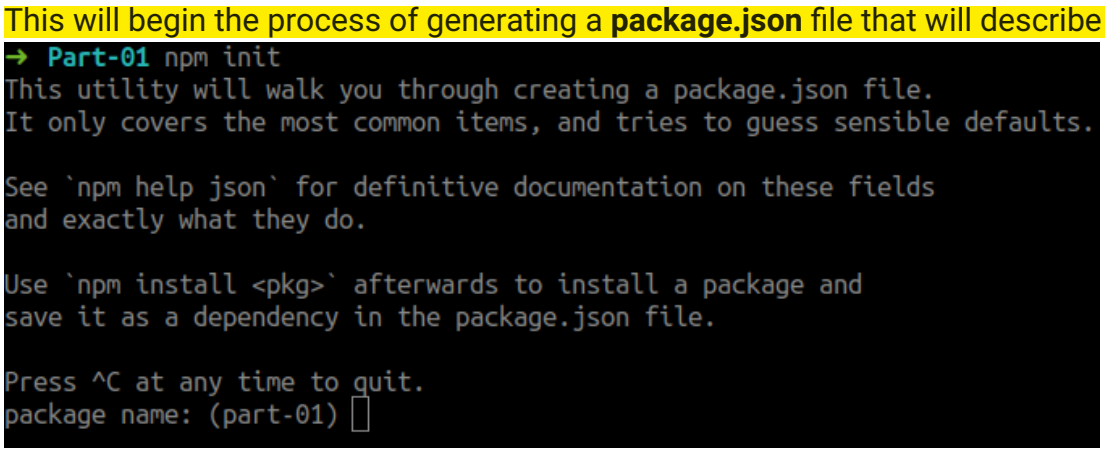
- Installed when you install NodeJS
- If you don't have NodeJs? Install NodeJS

Using NPM

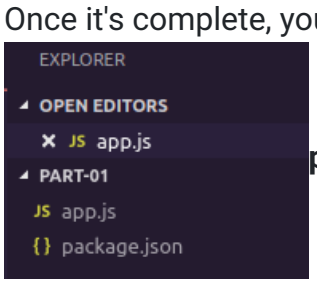
In your project directory in terminal/commandprompt/powershell

```
npm init
```

This will begin the process of generating a **package.json** file that will describe your application:



Once it's complete, you will have a new file in your project directory:



package.json describes your application. It provides a name for your application, the version number, your entry point file (**app.js**) and information about who created it. The **2 most imporant sections** are the **dependencies** and **scripts**

sections. The **scripts** section provides easy to use custom **CLI (command line interface)** commands that you can use to run tests, builds, or just the app in general. The **dependencies** list the **node modules** we have installed, and their information including their version numbers.

An example package.json file

```
{
  "name": "part-01",
  "version": "1.0.0",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {},
  "description": ""
}
```

When we first run **npm init** the **package.json** file will be quite sparse. As we install modules, our dependencies will grow. In addtition we'll also need to add a new script to run our application.

Installing a package using NPM

Once you find the right package, you'll be able to install it using the command

```
npm install <Package Unique Name>
```

Installing a module globally is similar to its local counterpart, but you'll have to add the **-g** flag as follows:

```
npm install -g <Package Unique Name>
```

For example, to locally install Express, you'll need to navigate to your application folder and issue the following command:

```
npm install express
```

The preceding command will install the latest stable version of the Express package in your local **node_modules** folder.

Furthermore, NPM supports a wide range of semantic versioning, so to install a specific version of a package, you can use the **npm install** command as follows;

```
npm install <Package Unique Name>@<Package Version>
```

For instance, to install the latest major version for the Express package, you'll need to issue the following command:

```
npm install express-current-version
```

Removing a package using NPM

To remove an installed package, you'll have to navigate to your application folder and run the following command:

```
npm uninstall <Package Unique Name>
```

NPM will then look for the package and try to remove it from the local **node_modules** folder.

To remove a global package, you'll need to use the **-g** flag as follows:

```
npm uninstall -g <Package Unique Name>
```

Updating a package using NPM

To update a package to its latest version, issue the following command:

```
npm update <Package Unique Name>
```

NPM will download and install the latest version of this package even if it doesn't exist yet.

To update a global package, use the following command:

```
npm update -g <Package Unique Name>
```

Installing the Package Dependencies

After creating your package.json file, you'll be able to install your application dependencies by navigating to your application's root folder and using the npm install command as follows:

```
npm install
```

NPM will automatically detect your package.json file and will install all your application dependencies, placing them under a local node_modules folder.

An alternative and sometimes better approach to install your dependencies is to use the following npm update command:

```
npm update
```

This will install any missing packages and will update all of your existing dependencies to their specified version.

A super simple Node server using Express

What is Middleware?

- Middleware glues together complex modular systems
- This could be
  - A messaging system
  - An event system
  - Task/Job scheduler
  - A router
  - An ORM or ODM
- Middleware can also be considered as a communication or interpreter system that allows output from one system to be work as acceptable input on another system

Why is Express Considered Middleware?

- Our user makes a request to our server which is listening
- Express intercepts the request and evaluates the request packet the user is sending
  - For example, Express evaluates the request path
  - From that, Express determines which registered middleware to execute
  - The interesting thing here is that the middleware is calling more middleware where we can perform operations based on the request packet
- The first intercept is known as routing

A Quick 2 Minutes on Express

- Express is a Node.js web server project
  - As stated earlier, the community is driving the web application side of Node.js
  - Express is one of those projects
- Express is a bit of sugar on top of Node.js' createServer
- Declarative routing
  - Routing is the process of analyzing, modifying, and delivering to the appropriate end-point

Building an Express Web Application

- The **Express framework** is a small set of common web application features, kept to a minimum in order to maintain the **Node.js** style.
- It is built on top of **Connect** and makes use of its middleware architecture.
- Its features extend **Connect** to allow a variety of common web applications' use cases, such as the inclusion of **modular HTML template engines**, extending the response object to support various data format outputs, a routing system, and much more.
- In order for us to use Express, we should initialize our package dependency file and install the Express module:

1. Create a new project folder on your computer called **BloggingProject**
2. Navigate (in your command prompt/terminal) to your **BloggingProject** folder
3. Once you are there, run the following commands:

```
npm init
npm install --save express
```

4. After creating your **package.json** file and installing express as one of your dependencies, you can now create your first **Express** application.
5. Open your **BloggingProject** folder in your **IDE**
6. Create a new file called **app.js**
7. Add the following lines:

Our very simple server with 2 routes/responses

```
// Our imported libraries
const express = require('express');

// Assigning Express to an app constant
const app = express();

// Creating our first route which is looking for requests from http://localhost:4000/
app.get(`/`, (req, res) => { // res is the response object and req is the request object
  // Our response
  res.send(`Home`);
});

// Creating our first route which is looking for requests from http://localhost:4000/
app.get(`/greeting`, (req, res) => {
  // Our response
  res.send(`Hey 'dere world!`);
});

// Starting our server on port 4000
app.listen(4000, () => console.log('Listening on 4000'));
```

Understanding Our Application

- **Express** presents three major objects that you'll frequently use.
  - The **application object** is the instance of an Express application you created in the first example and is usually used to configure your application.
  - The **request object** is a wrapper of Node's HTTP request object and is used to extract information about the currently handled HTTP request.
  - The **response object** is a wrapper of Node's HTTP response object and is used to set the response data and headers.
- The **application object** contains the following methods to help you configure your application:
  - **app.set(name, value)**: This is used to set environment variables that Express will use in its configuration.
  - **app.get(name):** This is used to get environment variables that Express is using in its configuration.
  - **app.engine(ext, callback):** This is 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:** This is used to send application-level variables to all rendered templates.
  - **app.use([path], callback):** This is 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):** This is 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.
  - **app.route(path).VERB([callback...], callback):** This is 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):** This is 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)**.

Deploying to Heroku

Signing up to Heroku

1. Navigate to https://signup.heroku.com/login
2. Fill in the information
3. Choose **'Node.js'** for the **'Primary Development Language'**
4. Click **'Create Free Account'**
5. Once you have confirmed your email and signed in:
    1. Click **Create new app**
    2. Name it your **lab-01-comp2068-STUDENTID**
    3. Click **Create app**
    4. Under the tab **Deploy**
        1. Scroll down till you see the section **Deploy using Heroku Git**
        2. We have a choice
            1. We can push everything to GitHub and deploy from there
            2. We can push to Heroku's repository and deploy from there
        3. We can always change our deploy method at a later date. For now, let's just use Heroku
    5. Navigate to https://devcenter.heroku.com/articles/heroku-cli
        1. Download and install for your OS
        2. Follow the installer instructions and install
        3. If you are in Windows
            1. Ensure **Set PATH to heroku** is checked
        4. Once you have it installed
            1. Close your CLI tool and reopen it
            2. Navigate to our **BloggingProject** folder
            3. In the command line follow along
                1. Login to heroku

                    $ heroku login

                2. Initialize your GIT

                    $ git init

                3. Pair up Heroku with your GIT

                    $ heroku git:remote -a lab-01-comp2068-STUDENTID

                4. Add your changes, commit, and push

                    $ git add .
                    $ git commit -am "Pushing for the first time."
                    $ git push heroku master

                5. Heroku will move your changes and build your environment for you automatically (if it can find a build)
                6. To open our application we need to run

                    $ heroku open

6. Your app didn't work
    1. Let's see why
        1. In the CLI you can output the Heroku errors from the log file

2018-05-06T19:45:50.796538+00:00 heroku[web.1]: Starting process with command `npm start`
2018-05-06T19:45:53.068278+00:00 app[web.1]:
2018-05-06T19:45:53.068292+00:00 app[web.1]: > helloworld@1.0.0 start /app
2018-05-06T19:45:53.068293+00:00 app[web.1]: > node express_server.js
2018-05-06T19:45:53.068295+00:00 app[web.1]:
2018-05-06T19:45:53.360734+00:00 app[web.1]: Server running at http://localhost:3000
2018-05-06T19:46:51.359500+00:00 heroku[web.1]: Error R10 (Boot timeout) -> Web process failed to bind to $PORT within 60 seconds of launch

2. So Heroku executed the script **npm start**
3. However, it looks like it blew up when it attempted to run the server on **localhost:4000**
4. The truth is we don't know what port Heroku wants to run on. In fact we don't really want to guess in case they ever change it. In addition, we want to be able to run it locally still so we can continue to develop.
    1. So how do get the best of both worlds?
    2. The wonderful || logical operator
    3. ```
       app.listen((process.env.PORT || 4000), () => console.log('Listening on 4000'));
       ```
    4. Basically this says if the **environment variable** is missing, use port **4000** instead
7. Let's try that one last time
    1. Add your changes, commit, and push

       $ git add .
       $ git commit -am "Pushing for the third time."
       $ git push heroku master

    2. Run **heroku open**

       Hello World

8. **YAY WE DID IT!!!**

Heroku

- We will require an app for everything we want to deploy
- Heroku is free unless you exceed their traffic limit, so don't
- Test in Development ALWAYS before deploying to Heroku
- Heroku is only a suggestion
    - You are welcome to deploy to where ever you choose
       ```

Understanding MVC

Deploying to Heroku

Signing up to Heroku

1. Navigate to https://signup.heroku.com/login
2. Fill in the information
3. Choose **'Node.js'** for the **'Primary Development Language'**
4. Click **'Create Free Account'**
5. Once you have confirmed your email and signed in:
    1. Click **Create new app**
    2. Name it your **lab-01-comp2068-STUDENTID**
    3. Click **Create app**
    4. Under the tab **Deploy**
        1. Scroll down till you see the section **Deploy using Heroku Git**
        2. We have a choice
            1. We can push everything to GitHub and deploy from there
            2. We can push to Heroku's repository and deploy from there
        3. We can always change our deploy method at a later date. For now, let's just use Heroku
    5. Navigate to https://devcenter.heroku.com/articles/heroku-cli
        1. Download and install for your OS
        2. Follow the installer instructions and install
        3. If you are in Windows
            1. Ensure **Set PATH to heroku** is checked
        4. Once you have it installed
            1. Close your CLI tool and reopen it
            2. Navigate to our **BloggingProject** folder
            3. In the command line follow along
                1. Login to heroku

                   $ heroku login

                2. Initialize your GIT

                   $ git init

                3. Pair up Heroku with your GIT

                   $ heroku git:remote -a lab-01-comp2068-STUDENTID

                4. Add your changes, commit, and push

                   $ git add .
                   $ git commit -am "Pushing for the first time."
                   $ git push heroku master

                5. Heroku will move your changes and build your environment for you automatically (if it can find a build)
                6. To open our application we need to run

                   $ heroku open

6. Your app didn't work
    1. Let's see why
        1. In the CLI you can output the Heroku errors from the log file

           2018-05-06T19:45:50.796538+00:00 heroku[web.1]: Starting process with command `npm start`

2018-05-06T19:45:53.068278+00:00 app[web.1]:
2018-05-06T19:45:53.068292+00:00 app[web.1]: > helloworld@1.0.0 start /app
2018-05-06T19:45:53.068293+00:00 app[web.1]: > node express_server.js
2018-05-06T19:45:53.068295+00:00 app[web.1]:
2018-05-06T19:45:53.360734+00:00 app[web.1]: Server running at http://localhost:3000
2018-05-06T19:46:51.359500+00:00 heroku[web.1]: Error R10 (Boot timeout) -> Web process failed to bind to $PORT within 60 seconds of launch

2. So Heroku executed the script **npm start**
3. However, it looks like it blew up when it attempted to run the server on **localhost:4000**
4. The truth is we don't know what port Heroku wants to run on. In fact we don't really want to guess in case they ever change it. In addition, we want to be able to run it locally still so we can continue to develop.
    1. So how do get the best of both worlds?
    2. The wonderful || logical operator
    3. ```
       app.listen((process.env.PORT || 4000), () => console.log('Listening on 4000'));
       ```
    4. Basically this says if the **environment variable** is missing, use port **4000** instead

7. Let's try that one last time
    1. Add your changes, commit, and push

       $ git add .
       $ git commit -am "Pushing for the third time."
       $ git push heroku master

    2. Run **heroku open**

       Hello World

8. **YAY WE DID IT!!!**

Heroku

- We will require an app for everything we want to deploy
- Heroku is free unless you exceed their traffic limit, so don't
- Test in Development ALWAYS before deploying to Heroku
- Heroku is only a suggestion
    - You are welcome to deploy to where ever you choose

QUIZ

BREAK

Routing In Depth

What is routing?

Routing is one of the most fundamental keys when working with MVC or Component based design patterns. Simply put, routing is the practice of interpreting a user's request and transferring it to the correct location. A very accurate real world example would be a phone operator. The operator takes a request from the person calling and connects them to the person they wish to reach. It uses the phone number in the user's request to correctly connect to the end point. **Routers** are known as **middleware** and are responsible for controlling traffic.

       ```

Introducing resource routing

Resource routing is a design pattern that utilizes a naming convention to define what action a users request will result in. For example, if a user attempts to access **https://amazon.ca/products** the router will direct the request to the **index action**. If they attempt to access **https://amazon.ca/products/12345** the router will direct the request to the **show action**.

The router looks at both the **header** and the **message body** of the request to determine what action it needs to execute. The **header** will provide 2 key pieces of information: **the path** the client requested and **the method** they used to make the request. By using both sources of information we can have identical route paths that execute unique operations. You can see this if you observe the table below. If a user attempts to **GET** the path **/resources** they will execute the **index action**. However, if they **POST** to the path **/resources** they will execute the **create action**.

Method	User Request	Intercepting Route	Controller Action
GET	/resources	/resources	index
GET	/resources/12345	/resources/:id	show
GET	/resources/new	/resources/new	new
GET	/resources/12345/edit	/resources/:id/edit	edit
POST	/resources	/resources	create
POST	/resources/12345	/resources/:id	update
POST	/resources/12345/delete	/resources/:id/delete	delete

Treating Pages as a Resource

In order to better understand **resource routing** we're going to treat common pages as a resource. The pages we'll create will be **Home**, **About**, and **Contact**. Our first step will be to create some **route rules** that will tell the router what actions to execute. Let's create some new folders and files. This will be our new file and folder structure (new structures are in blue):

- /
  - node_modules
  - **routes**
    - **pages.js**
  - app.js
  - **routes.js**
  - package.json
  - package-lock.json

Moving our current routes from app.js

We're going to move our current routes from **app.js** into our new file pages**.js**. We're doing this to maintain modularity. It will be easier for us to add new pages to our application by simply opening our pages.js file and following our logic. Before we can add the routes though, we need to **require** the **Express Router** module. If you look at our code below you will see the following line:

```
// Our router module
const router = require('express').Router();
```

This line is not only requiring the **express** library but immediately invoking and returning the **Router() method** and assigning it to our variable **router**. So that means that **const router** is now equal to whatever **Router() returned**. This is identical to doing this:

```
// Our router module
const express = require('express');
const router = express.Router();
```

pages.js

```
// Our router module
const router = require('express').Router();

// Creating our first route which is looking for requests from http://localhost:4000/
router.get(`/`, (req, res) => {
  // Our response
  res.send(`Home`);
});

// Creating our first route which is looking for requests from http://localhost:4000/
router.get(`/about`, (req, res) => {
  // Our response
  res.send(`Hey 'dere world!`);
});

// We have to export our changes
module.exports = router;
```

The syntax **module.exports** is a little new for us. **module** is a global object that Node creates when it runs. The module object contains many properties and functions that you can tap into to use for different operations. One of those is **exports.** The **exports** property provides us a way to bind our current logic state and pass it to the **requiring** file. For example, if I'm requiring **pages.js in routes.js** the **router** that I'm exporting will be what is required (open routes.js and write the logic below):

routes.js

```
// Our Express app module
const express = require('express');
const app = express();

// Importing the pageRoutes
const pageRoutes = require('./routes/pageRoutes');

// Registering our pageRoutes
app.use('/', pageRoutes);

// Exporting the changes
module.exports = app;
```

The **router** we exported in **pages.js** is now assigned to the **pageRoutes** variable. It contains all the routes we registered for pages. Now we need to get this back into our application so it's registered. In order to do that, we're going to have to once again **export**, but this time we'll be exporting the **app**.

NOTE: We can definitely simplify this code, but for readability I feel this is much simpler to understand.

Lastly, we want to modify our **app.js** to import the new router.

app.js

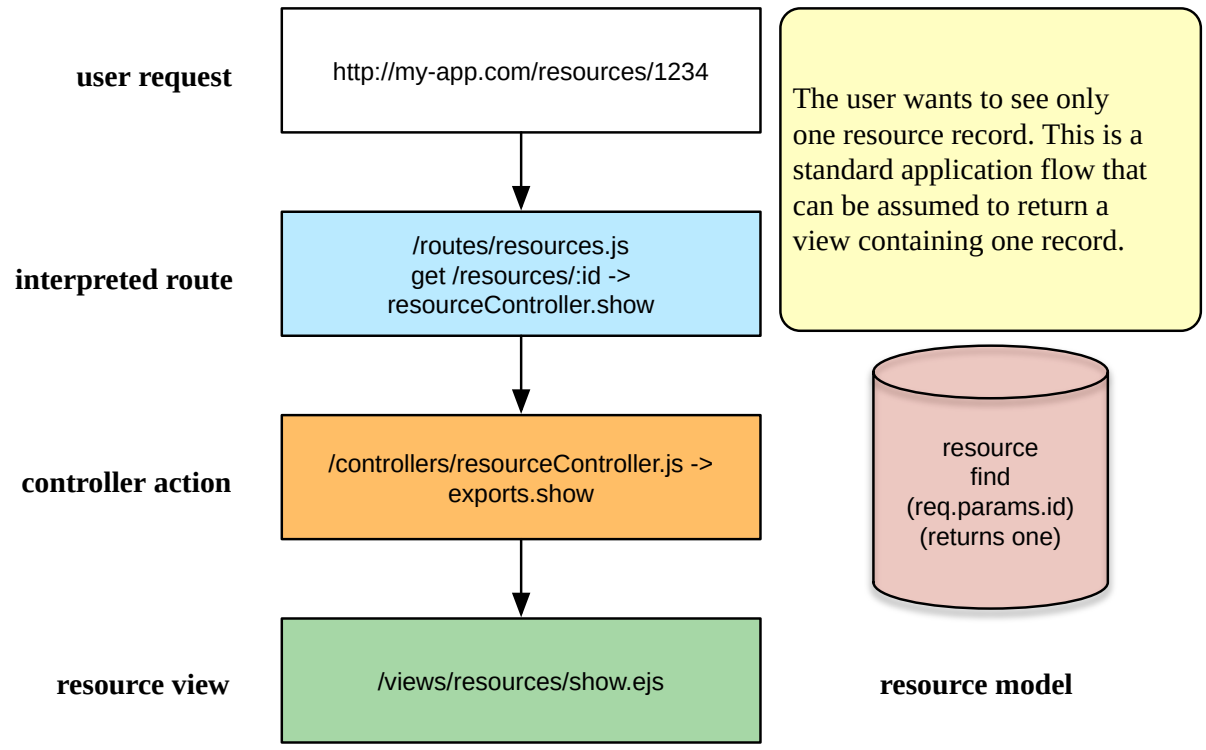
```
// Our imported libraries
const express = require('express');

// Assigning Express to an app contstant
const app = express();

// Our routes
const routes = require('./routes.js');
app.use('/', routes);

// Starting our server on port 4000
app.listen((process.env.PORT || 4000), () => console.log('Listening on 4000'));
```

Introducing MVC and Our Folder Architecture

The Express (Resource Routes) Application Flow

Use Case:

I want to see one resource record:



We don't have a model resource in our current structure, but we do have a resource. Our resource is our **page**. We're going to have 3 resources in total: **home**, **about**, and **contact**. These resources will all go through one controller action and the appropriate view will be returned as a response. The controller action (based on our convention) that makes the most sense is **show**. The nice thing is we will only have the one controller action in the **pagesController**.

In order to add the controller and the views, we'll need some new files and folders:

- /
  - node_modules
  - routes
    - pageRoutes.js
  - **controllers**
    - **pagesController.js**
  - **views**
    - **pages**
      - **home.pug**
      - **about.pug**
      - **contact.pug**
  - app.js
  - routes.js
  - package.json
  - package-lock.json

Understanding Our Folder Structures

- /
  - **/views** *(our resource **views** go here)*
    - **/pages**
      - home.pug
      - about.pug
      - contact.pug
  - **/controllers** *(our resource **controllers** will go here)*
    - pagesController.js *(this is our resource controller file that defines the actions each view will need)*
      - show/:pageName *(this method will **fetch one** page and **render the show.pug view**)*
  - **/routes** *(our resource **routes** will go here)*
    - pages.js *(this is the defined routes for our resource)*
      - **get** / -> pagesController.show *(this route will pass the **request** body to the **index** method in the **pagesController**)*
      - **get** /about -> pagesController.show *(this route will pass the **request** body to the **show** method in the **pagesController**)*
      - **get** /contact -> pagesController.show *(this route will pass the **request** body to the **show** method in the **pagesController**)*

Prepare for Views

Preparing app.js

In order to accommodate views we need to update our **app.js** file with a **views path** and a **view engine**. The **view path** is a default location for the app to look for a valid view to **render**. The **view engine** is what type of templating engine we want to use in order to create our views. We can use basic HTML, EJS, Jade, HAML, or (the one I chose) **pug**. **Pug** is nice because it's a **tab-delimited templating language** that allows for the construction of web pages to be super fast and efficient. It supports inline JavaScript (essential for our application) and a whole slew of helpful utilities for doing conditional logic and iterations.

Below you can see the modifications made. I added the following 5 lines:

```
// This maintains our home path
const path = require('path');

// Our Views
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
```

The first line requires the **path module**. The **path module** will allow us to interact with our application's path structure more easily. Next we **set our view path** by using the path module to join **views** to our path structure. Last we **set our view engine**. The view engine we'll be using is **pug** which also means we'll need the **pug module**.

app.js

```
// Our imported libraries
const express = require('express');

// Assigning Express to an app contstant
const app = express();

// This maintains our home path
const path = require('path');

// Our Views
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');

// Our routes
const routes = require('./routes.js');
app.use('/', routes);

// Starting our server on port 4000
app.listen((process.env.PORT || 4000), () => console.log('Listening on 4000'));
```

Installing the Pug and Nodemon Modules

Not much to installing the **Pug module**. We install it the same way we install any module in our application:

```
node install pug --save
```

While we're at it, why don't we install **Nodemon**. This module makes running and making changes in our application very easy as it doesn't require us to restart after every change. This will be our first **global node module**, but installing it is still very simple:

```
node install nodemon -g
```

The **-g** flag will tell node we want to install this module at the global level. Now we will be able to run nodemon in any application that contains a package.json file. It will look for the **entry file** in our package.json file and execute it. To use nodemon, simply run

```
nodemon
```

in your terminal.

BREAK

Controllers

What are Controllers?

Controllers are a collection of actions that handle request bodies and provide responses. Their primary function is to facilitate transferring data from the model to the view, or requests from the view to the model. Like the router, they essentially work as a middleware. In fact, the router and the controller are the perfect marriage of technologies that handle all request/response traffic. Every request will be handled with a controller. All responses will be delivered from a controller.

Our Pages Controller

The first controller we're going to create will be the **PagesController**. Controllers are made up of possible actions they can perform. They may also contain utility operations that pertain specifically to the controller. For now, our controller is going to be very simple and will only contain one action: **show**.

/controllers/PagesController.js

```
// Our controller action
exports.show = (req, res) => {
  // Check if our path is the root or a page
  const path = (req.path === '/') ? '/home' : req.path;

  // Render that path
  res.render(`pages${path}`);
}
```

Our controller action is made up of a few key pieces:

- **exports.show** - this is the name of our controller action. In more model based controllers we'll have all the resource route actions: index, show, new, edit, create, update, and delete
- **(req, res) => {}** - this is an anonymous arrow function with 2 parameters: **req**uest and **res**ponse. When our route executes our function it will pass in two objects to fill these parameters as arguments. The **request object** will contain a lot of information about the request the user sent. The **response object** is our way to communicate back to the user. We have many properties and methods on both objects to use, but we'll only be using a few in our project
- **(req.path === '/') ? '/home' : req.path;** - this is a ternary. We're checking the value of the **req.path** property and seeing if it is the **root path** or not. If it is the root path, then we deliver the **'/home'** path otherwise we use the existing path. This will help us to dynamically determine the view we want to respond with
- **res.render(`pages${path}`);** - **res.render** is a method available on the **response object** that gives us a way to respond with a view. We simply point where our view is and it will render it using the defined **template engine**
    - Incidentally, **`pages${path}`** will evaluate to **pages/home** or **pages/about** or **pages/contact**
    - The application will look for our path under **/views** and append the path provided to the **res.render()** method. This means **res.render()** will be looking in **/views/pages** for our current views

---

Attaching PagesController to our Router

In order to direct traffic to the correct action, we'll need to add the **PagesController** to our router. We can do this by simply requiring the PagesController in our /routes/pages.js file and setting it as the callback function:

/routes/pages.js

```
// Our router module
const router = require('express').Router();

// Our controller
const PagesController = require('../controllers/pagesController');

// Our routes
router.get(`/`, PagesController.show);
router.get(`/about`, PagesController.show);
router.get(`/contact`, PagesController.show);

// We have to export our changes
module.exports = router;
```

In the above we've replaced the anonymous callback functions with our new logic. Each route is defined and will explicitly call our action on the **PagesController**. In most of our resource files we'll only ever have 7 routes, but the logic is the same. This modular approach will make working in our application a pleasure as everything won't be a tangled mess of spaghetti code.

Views

Pug

https://pugjs.org/api/getting-started.html

Pug is a template language used to write HTML in a faster and more efficient way. Writing pug is quite simple. You don't need <> anymore. You don't need closing tags either:

simple pug example

```
head
  head
    title My Awesome Title
  body
    h1#header Best Site EVER!!!

    p.intro-text(data-attr="my value")
      It was a rainy day in pizzaville...
```

The tags are now reduced to one word that prefixes whatever you value will be. #ids are used to define a tag with an ID. .class-name syntax is used to define a CSS style class. Any other attributes are encased in the following parenthesis. Over the next 4 weeks we'll be using Pug a lot so you will see all the little utilities it provides.

Layouts

Many frameworks support **layouts**. The idea of a layout is to create a template that you can drop a body into. If you consider an HTML page, there are many pieces that are simply duplicated from page to page. The HTML doctype and declaration, the HTML, HEAD, and BODY tags, the title tag... These can be handled in a layout using the very handy **block** utility built into **Pug**.

The **block utility method** essentially puts in a placeholder that the **extending view** can populate. For example:

example layout

```
doctype html
html(lang="en")
  head
    title My Home Page

  body
    block content
```

block content is our placeholder, **content** being the placeholder's name. When the template is parsed, it will replace **block content** with the extended files content:

example extended content

```
extends ../layouts/main.pug

block content
  header
    h1 My Home Page
```

The content **nested** in **block content** will replace the **block content** declaration in the **example layout**. It is important to remember to **extend the layout** using the command **extends ../path/to/layout.pug**.

Let's use the above example to create a **main.pug** layout that we can use to wrap our content:

/views/layouts/main.pug

```
doctype html
html(lang="en")
  head
    title= (title ? title : "I do not have a title yet!")

  body
    include ../partials/main-nav.pug

    block content
```

You'll notice we added a line **include ../partials/main-nav.pug**. This line will include a file as a **partial view**.

Partials

Partials help us reduce redundant views. We can use them to populate small repetitious pieces of content. For example, we'll want a nav menu in our views so we can easily move around our application. We could code this in our **main.pug** file to reduce repetition, but later on we're going to want to change our menu based on who's logged in and what their role is. We can deliver different navigation menus based on that logic, but they'll need to be in their own files. To do this, we simply create a new pug file and place it somewhere in our app. Then we populate it with our HTML structure:

/views/partials/main-nav.pug

```
ul
  li
    a(href='/') Home
  li
    a(href='/about') About
  li
    a(href='/contact') Contact
```

Once we've done that, we simply add an **include** line wherever we want our navigation to show:

```
include ../partials/main-nav.pug
```

We already did this in our **layouts/main.pug** file.

Home Page

For now we'll create some very simple pages. We already created the 3 view files, now we'll just populate a basic title in message in each file.

CHALLENGE! We have a slight issue: We will want to populate the title of the page in the **head > title** tag. But our Home page is extending the main layout. If we look at how our code will process we'll see that our **home page** is copied and embedded into our **main.pug** file, replacing our **block content** declaration. If we were to set a variable in our home page file, it would come after the

title tag, meaning the variable would never be able to be used. The good news is we can utilize the **block** placeholder utility once again to define a set of variables in the right location. Let's first modify our **main.pug** to have a new block:

/views/layouts/main.pug

```
block globals

doctype html
html(lang="en")
  head
    title= (title ? title : "I do not have a title yet!")

  body
    include ../partials/main-nav.pug

    block content
```

We'll use **block globals** as our placeholder for any declarations we want to make before our HTML logic. Next we'll add a **block globals** to our home.pug file and define a variable in it to hold the title:

/views/pages/home.pug

```
extends ../layouts/main.pug
block globals
  - title = "My Blogging Platform"

block content

  header
    h1= title
```

The remainder of the file extends the layout and adds the content we want to pass to **block content**. We can now easily pass variables to the layout and our content to the layout.

About Page

The about page will just duplicate the home page logic for now. Next lesson we'll add our CSS and flesh these pages out more.

/views/pages/about.pug

```
extends ../layouts/main.pug
block globals
  - title = "About My Blogging Platform"

block content

  header
    h1= title
```

Contact Page

We will add a map to this page as well by adding a way to serve static JavaScript files. We'll look at that next week as well.

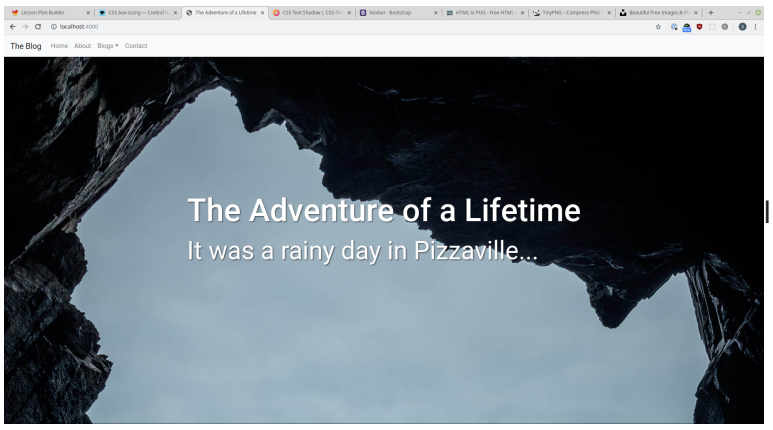
/views/pages/contact.pug

```
extends ../layouts/main.pug

block globals
  - title = "Contact Me on My Blogging Platform"

block content

  header
    h1= title
```

Models & MongoDB Part 1

A Better Home Page

Home Page Changes



I've modified the Home Page (**views/pages/home.pug**) to have a nice modern feel. I got the background image from Unsplash, a site that provides free stock photos. The image is a coastal area that I've cropped to work for our background. In all honesty, I haven't changed our view all that much. The real power is the CSS. However, our app doesn't currently support assets (CSS, JavaScript, and images) so we'll have to remedy that.

Place the code below in the appropriate files. We'll need a new directory structure to add the CSS and the image:

- **assets**
  - **images**
  - **javascripts**
  - **stylesheets**
    - **styles.css**
- controllers
- routes
- views
- app.js
- routes.js
- package.json

The Code

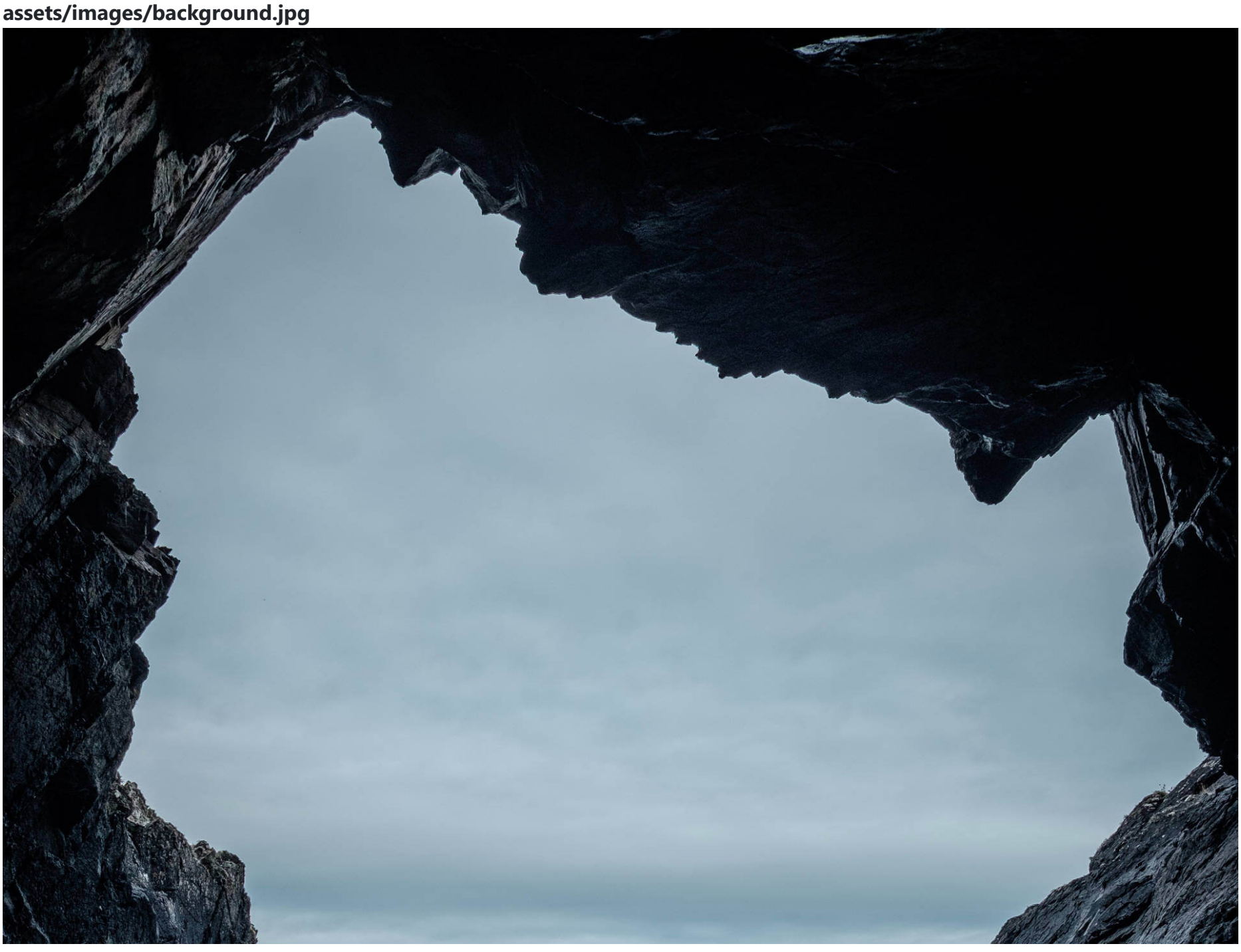
views/pages/home.pug

```
extends ../layouts/main.pug

block globals
  - title = "The Adventure of a Lifetime"

block content

  header.home-cta
    h1
      = title
      br
      small It was a rainy day in Pizzaville...
```

assets/stylesheets/styles.css

```
html,
body {
  height: 100%;
  width: 100%;
}

/* Photo by Marco Molitor on Unsplash */
.home-cta {
  background: url('/images/background.jpg');
  background-size: cover;
  height: 100%;
  width: 100%;
  display: flex;
  justify-content: center;
  align-items: center;
}

.home-cta>h1 {
  color: #fff;
  text-shadow: 2px 2px 3px rgba(0, 0, 0, 0.5);
  font-size: 5rem;
}

.content {
  margin-top: 60px;
}
```

assets/images/background.jpg



Adding the Asset Pipeline

Asset Pipeline

An asset pipeline is usually a directory structure (that isn't public) that you place your assets (CSS, JavaScript, and images) into. The pipeline will usually have added functionality like compression, minification, uglification, optimization, transpiling, etc... that will immediately be done when an asset is accessed. In addition, the pipeline will provide convenient public facing routes to our assets that make accessing them a breeze for our clients.

In order to create an asset pipeline in our Node application we will need to modify our **app.js** file and register some new paths for CSS, JavaScript, and images:

app.js

```
// Our Views
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
app.use('/css', express.static('assets/stylesheets'));
app.use('/js', express.static('assets/javascripts'));
app.use('/images', express.static('assets/images'));
```

app.use is a method provided to us by Express that allows us to register a middleware action to a route. In this scenario, we are registering the paths **'/css'**, **'/js'**, and **'/images'** to point to our asset directory and their corresponding folders. Because the second parameter to **use** can be a callback, we could provide more sophisticated logic that would perform extra operations on our assets such as the ones I listed before (compression, minification, etc...).

Adding Our styles.css to Our Application

We have a stylesheet that we created and populated. It isn't currently being applied to our application and we will want to rectify that. While we're at it, we may as well also add in some external frameworks to help us beautify our app. All of this logic is going to be applied to our wonderful **views/layouts/main.pug** file:

views/layouts/main.pug

```
block globals

doctype html
html(lang="en")
  head
    link(href="/css/styles.css", rel="stylesheet")

    title= (title ? title : "I do not have a title yet!")

  body
    include ../partials/main-nav.pug

    block content
```

In the above code block we've made one change; we've added the link to our stylesheet. Notice the link doesn't point to **assets/stylesheets/styles.css** but rather the registered path we set in our **app.js**. This is a cleaner approach.

Now we're ready to add 2 external frameworks we'll need for our application. Those will be **Twitter's Bootstrap** and **JQuery**. Both of these will be served from their CDN (Content Delivery Network). We won't be using JQuery to it's fullest as we have the helpful hand of ES6+. However, Bootstrap requires JQuery for many of its functional components, so we'll need to make sure it's present.

views/layouts/main.pug

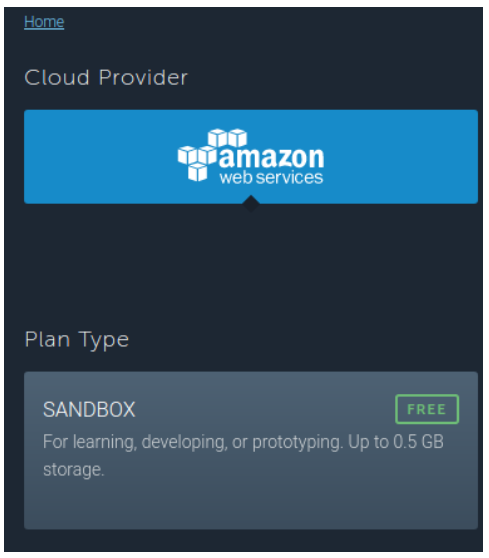
```
block globals

doctype html
html(lang="en")
  head
    link(href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css", rel="stylesheet",
integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T", crossorigin="anonymous")
    link(href="/css/styles.css", rel="stylesheet")

    title= (title ? title : "I do not have a title yet!")

  body
    include ../partials/main-nav.pug

    block content

    script(src='https://code.jquery.com/jquery-3.4.1.min.js' integrity='sha256-
CSXorXvZcTkaix6Yvo6HppcZGetbYMGWSFlBw8HfCJo=' crossorigin='anonymous')
    script(src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.bundle.min.js", integrity="sha384-
xrRywqdh3PHs8keKZN+8zzc5TX0GRTLCcmivcbNJWm2rs5C8PRhcEn3czEjhAO9o", crossorigin="anonymous")
```

First we place the Boostrap CDN link above our stylesheet. That way our stylesheet can override specific properties in the Bootstrap framework if we choose to. Next we add in the JQuery CDN dependency for Bootstrap. Lastly we add the Bootstrap CDN which will provide component functionality to the framework. One of those components will be our navigation's dropdown menus.

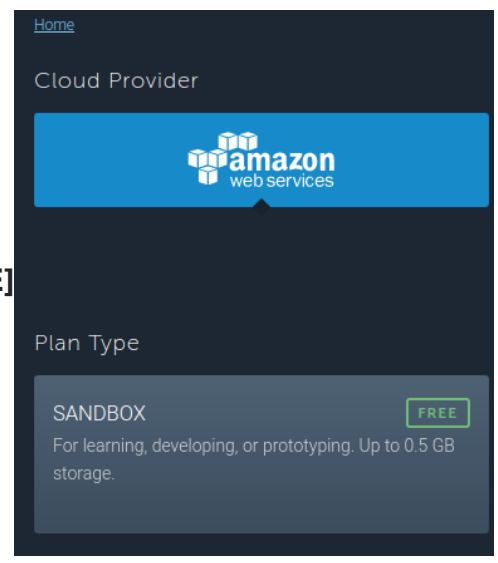
Creating an MLab Account & DB and Environment Variables

Setting up Your MLab Account

1. Go to MLab and create an account (click the **GET STARTED INSTANTLY**) button
2. Go to your email and confirm
3. Login
4. Click on the **Create New** (under the MongoDB Deployments) to create a new Mongo database

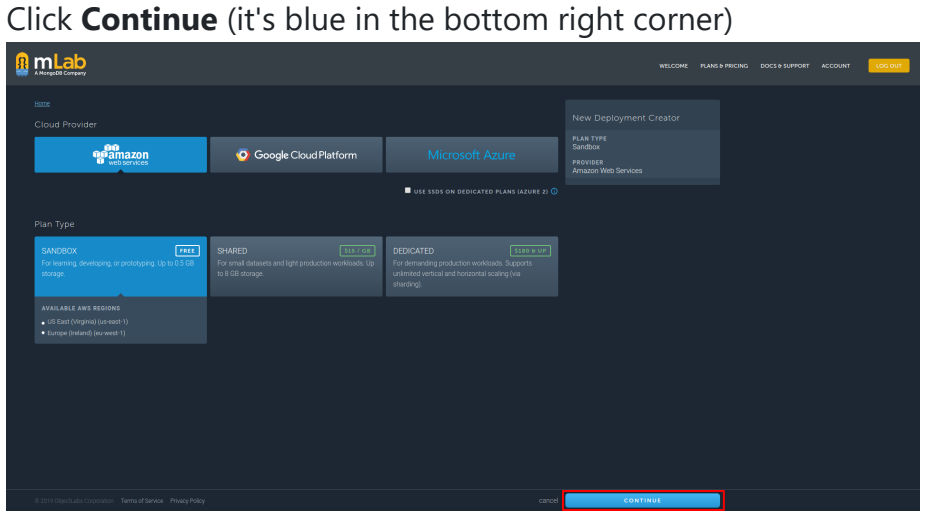


5. Choose **Amazon Web Services**

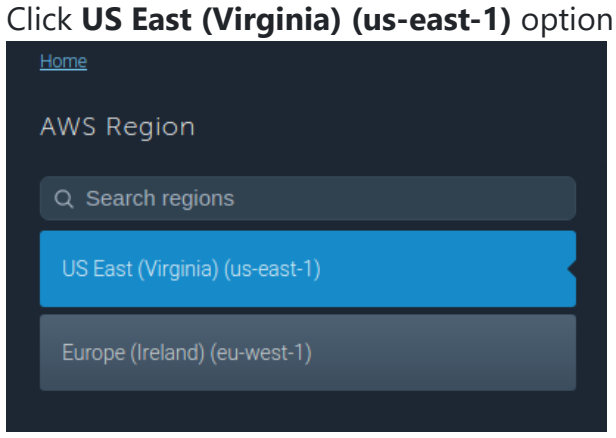


6. Choose **SANDBOX [FREE]**

7. Click **Continue** (it's blue in the bottom right corner)



8. Click **US East (Virginia) (us-east-1)** option



9. Click **Continue**
10. Type in **comp2068** (or whatever name you want to give your database)
11. Click **Continue**
12. Click **Submit Order**
13. You will see a table with your database name listed in it: click on it
14. Click on the tab **Users**
15. Click **Add database user**
16. Fill in the details to create a new user

Environment Variables

Environment variables are variables that exist only on your computer and are registered to your user. These variables are awesome because they're very secure and difficult to get for a hacker. Plus they are the preferred way of working with security credentials in situations where you're deploying to a production system. Our development credentials will often be different than our production ones, so it's nice to be able to have those credentials stored in environment specific locations.

We now need to configure our application to use MLab. We want to make sure we do this in a way that our credentials aren't being exposed to the internet in our repository. To do that, we're going to use an **environment variable** manager called **.dotenv**. It will allow us to create some local environment variables that will be used for the credentials. We can then do the same in our Heroku instance when we're ready to deploy.

First we'll need to install .dotenv. Open your terminal and type the following command:

```
npm install dotenv
```

Next we will need to create a new file at the root of our application:

- assets
- controllers
- views
- app.js
- routes.js
- **.env**
- .gitignore

The **.env** file has no file extension (just like the .gitignore file). Next we'll need to add **.env** to our **.gitignore** so it doesn't get picked up when we push our repository to GitHub:

```
node_modules
.env
```

Lastly we need to add some environment variables to our **.env** file:

```
DB_URI="mongodb://ds456123.mlab.com:4000/databasename"
USERNAME="yourdatabaseusername"
PASSWORD="yourdatabasepassword"
```

Your DB_URI will be different than the one above (and obviously your database credentials). You can find out what your URI here in MLab:

BREAK

SQL VS NoSQL

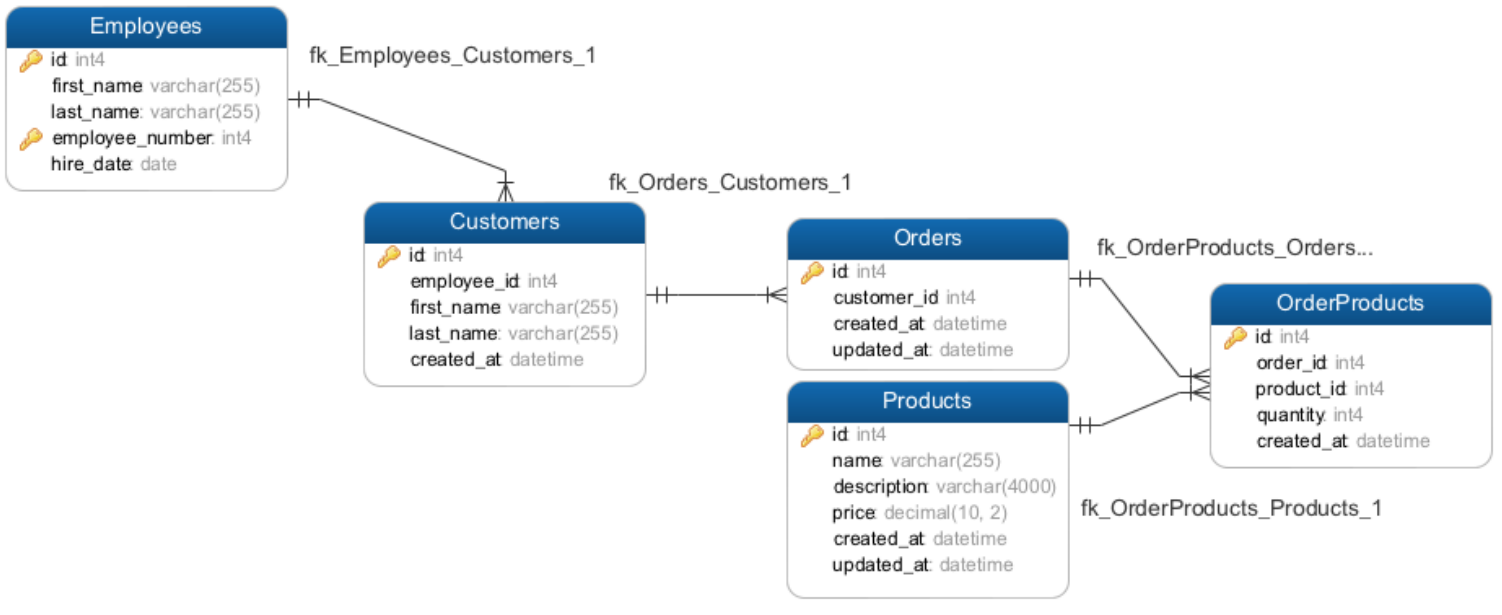
https://medium.com/chingu/an-overview-of-mongodb-mongoose-b980858a8994

What is a database?

There are several different types of databases, but they share one fundamental purpose: to store information. There are 3 fundamental types of databases that you will likely see in your quest for work: SQL, NoSQL, and Distributed File System. The latter one is less common, but if you're interested I would recommend Googling Hadoop. The other two are quite common and the subject of debates, as some appreciate the structure and relationships of SQL, while others like the flexibility and potential scalability of NoSQL.

What is so great about SQL (Relational DBs)

In the world of business, business people love to see structures that represent their business. **Customers**, **Employees**, **Orders**, **Products**, etc. In a SQL DB these would be likely tables, and they represent business data related to their names. In addition, we can visualize the relationships between these tables:



This structure will allow us to view all the customers under a specific employee. In addition we can see all the orders a customer has made and what products they had ordered. From that we can build analytical data on what products are more popular; what products are purchased in certain area; which employees have more active customers; which employees have signed the most customers.

SQL gives us several queries that allow us to control how we interact with data. Because there are so many different SQL databases, we can't be sure what queries we have available to us, as well as what datatypes there are available. This leads into the need for an ORM (Object Relational Mapper) layer. This layer abstracts our database into objects with attributes, allowing us to disregard data-typing at the database level. For example, MySQL has Enum and PostgreSQL does not. The ORM, when interacting with MySQL, will store enum values as enum strings. The ORM, when interacting with PostgreSQL, will instead store the values as integers. How you implement enum will be based on the ORM's requirements. This means that you learn the ORM and not the database.

What is so great about NoSQL DBs?

Because SQL relies on data relationships, this requires us to plan our database requirements in depth, in advance. Scalability becomes more difficult as new data requirements require us to migrate old data to new schemas. I'm sure you can see the potential problems here: missing data, corrupted data, improperly copied data, dropped relationships, etc... Scaling strict relational databases requires a lot of planning to avoid the above issues as well as money to pay for that planning. Often we will implement redundancies so we can rollback in case any errors occur. We will spin up entire redundant applications in order to ensure there is no downtime in case of an error.

NoSQL lowers the need for redundancy and planning. You can add new schema without affecting the old easily and in place without damaging your application. Because it's document store, your database can horizontally scale across new systems. SQL requires more resources in order to scale.

When should I use what?

Let's use some scenarios to determine the best choice:

I need an application to store an Organizational Hierarchy:

Requirements
A table for employees information

- name
- employee number
- position_id

A table for the organizations positions

- title
- parent position_id

My schema is fairly fixed. My employee information will be based on my business needs. The position table will allow me to build a hierarchy easily, and not require any flexibility. My data requires integrity to maintain its structure. A SQL database is made for this, and therefore the best choice for this scenario.

Let's look at a different scenario:

I need an application to store a Catalogue of Products:

Requirements
A table to store a product:
BUT WAIT! Some products have specifications. Some have configurations. Some have features. Some have benefits. Some have minimum purchase amounts. Some of maximum purchase amounts. Some become cheaper based on quantity.

A table to store a product:

- name
- description
- msrp

A table to store specifications
A table to store configurations
A table to store features
A table to store benefits

- type (spec, config, feat, or ben)
- product_id
- label
- description

That doesn't work:
For this specific spec I have images and tables for it. For this configuration, I want it to affect the price because it requires more material. This feature is tabular data, but this feature is bullet pointed.....F***!

In case you think this didn't happen: http://www.asbheat.com/products
We used a relational database for the mess described above. Our hierarchy was a nightmare and eventually we gave up. We created a catch all table and coded out each section individually based on its requirements. We essentially did document storage in a relational database.

A NoSQLsolution:

```
{
  name: '',
  msrp: '',
  details: {
    specifications: {
    },
    features: {
      tabbular: {
        tab1: '',
        tab2: '',
        tab3: ''
      },
      table: {
        header: [],
        body: []
      }
    }
  },
  price_affects: {
    quantity: {
      less: {
        amount: 15,
        mutation: {
          increment: 10,
          type: 'percent
        }
      },
      more: {
        amount: 100,
        mutation: {
          decrement: 20,
          type: 'percent
        }
      }
    }
  }
}
```

Many large cart systems utilize relational databases. Products are one of those beasts that will shift and change. Think of even product specific stores like a car dealership. Yes, every car will have tires, body type, and colour, but they will also have other details that affect specific types of cars. A utility van and a passenger car are not the same when we consider features and benefits. NoSQL allows us to use unstructured data and search that data efficiently.

One note: many companies are starting to see the need for non-hierarchical data structures. These same companies are having difficulty changing into systems like MongoDB. Instead they're introducing JSON and JSONB columns into their DBs, or they're adding 'tagging' systems that allow them to group data dynamically. Always remember, **right tool for the job**.

Mongoose & Models

What is Mongoose?

Mongoose is an ODM, which stands for **O**bject **D**ocument **M**apper. Basically the idea is to make writing validation, sanitization, business logic, and other boilerplate code simpler and faster. Mongoose will become our application's models, which will allow us

to abstract our data logic from our view.

Adding Mongoose to Our App

We add Mongoose the same way we do any package:

```
npm install mongoose
```

Next, open **app.js** and add the following lines at the top of the file (**below where we requires .dotenv**):

```
// Connecting to MongoDB cluster with Mongoose
const mongoose = require('mongoose');
mongoose.connect(process.env.DB_URI, {
  auth: {
    user: process.env.USERNAME,
    password: process.env.PASSWORD
  },
  useNewUrlParser: true
}).catch(err => console.error(`ERROR: ${err}`));
```

If everything is working, we should be able to start our application and we shouldn't get an error.

Creating a New Resource

MVC relies on separating data from presentation from logic. We use Models for our data, Views for our presentation, and Controllers for our logic. The benefit of this structure is it lends to better scalability, maintainability, and stability.

Models are our data piece, and they're great in that they encapsulate sanitization, validation, and mutation all in one area. The operations we perform on data should happen here, as the controller should simply facilitate the transfer of the data to an endpoint (perhaps the view). In our application, we're using Mongoose to help us create models and interact with our database.

First we're going to need a **models** directory in our application and we may as well create our first model filed too, **blog.js**:

- assets
- controllers
- **models**
    - **blog.js**
- views
- routes
- app.js
- ...

Blog will be our **resource** which means we're going to need the same components we used with **pages**. We'll need routes, views, and a controller to put this resource together and give it exposure to our users:

- assets
- controllers
    - **blogsController.js**
    - pagesController.js
- models
    - blog.js
- views
    - **blogs**
        - **index.js** (will list all of our blog posts)
        - **show.js** (will show one blog post)
        - **new.js** (will provide a form for us to create a new blog post)
- routes
    - **blogs.js**
    - pages.js
- app.js
- ...

The Resource Routes

We'll create the resource routes first. We will eventually have all 7 of the necessary resource routes, but for now we're going to just create 4:

routes/blogs.js

```
// Our router module
const router = require('express').Router();

// Our controller
const BlogsController = require('../controllers/blogsController');

// Our routes
router.get(`/`, BlogsController.index);
router.get(`/new`, BlogsController.new);
router.get(`/:id`, BlogsController.show);
router.post(`/`, BlogsController.create);

// We have to export our changes
module.exports = router;
```

The above code should seem familiar; it's identical to the **routes/pages.js** file we created last week. Here we are listening to 4 routes:

- **get '/'** - *which will return a list of blog posts* (**index**)
- **get '/new'** - *which will return a form for us to create a new blog post* (**new**)
- **get '/:id'** - *which will return a single specific blog post* (**show**)
- **post '/'** - *which will send the form data for a new blog post to be created* (**create**)

It isn't accidental that **/new** comes before **/:id**. **/:id** is known as a **bound parameter**. Bound parameters are placeholders (similar to function parameters) that will capture information. For example, if navigate to **/blogs/12345** then the bound parameter **:id** will hold the value **12345**. This becomes available to me in my controller and I can use it to lookup that specific resource. You can have as many bound parameters as you want and they can also follow specific rules. The reason **/new** has to come before **/:id** is because the router will confuse **new** as an argument for the bound parameter, which we don't want. This will cause a logic error when we attempt to lookup the specific resource.

IMPORTANT: Routes are verified in the order they're defined which means our requested path will be run past each route in the order they're typed in our routes file. This does get filtered down by the HTTP method.

We are almost done but we still need to register the new resource routes with our application. We do this in our **routes.js** file we created last week:

routes.js

```
// Our Express app module
const express = require('express');
const app = express();

// Importing the pageRoutes
const pageRoutes = require('./routes/pages');
const blogRoutes = require('./routes/blogs');

// Registering our pageRoutes
app.use('/', pageRoutes);
app.use('/blogs', blogRoutes);

// Exporting the changes
module.exports = app;
```

We just simply add the new routes under our pageRoutes. One slight change, because we want to constrain our routes to the resource type, we can bind the blog routes to all be prefixed with **/blogs**. This means our routes will be as follows:

- **get '/blogs/'**
- **get '/blogs/new'**

- **get '/blogs/:id'**
- **post '/blogs'**

The Mongoose Model

https://mongoosejs.com/

Mongoose makes creating models very easy. Within the model we can define our fields, the data type we expect, the validation we want to occur, and any methods we want as helpers. Below is an example of a model:

```
// We will need our mongoose library
const mongoose = require(`mongoose`);

// Our schema
const ModelnameSchema = new mongoose.Schema({
  field1: { // field name
    type: String, // data type
    required: true // validation
  },
  field2: { // field name
    type: Number, // field type
    required: false // validation
  },
  field3: { // field name
    type: String, // field type
    enum: ['val1', 'val2'], // enum values (restricted possible values)
    default: 'val1' // default value to be chosen
  }
}, { // additional options to be set on the model
  timestamps: true // default timestamps (createdAt and updatedAt)
});

// Query Helper
ModelnameSchema.query.helperMethod = function () { // a method to help return a specific set of documents
  return this.where({ // a query method that Mongoose provides to find documents
    status: 'val1'
  });
};

// Exporting our model
module.exports = mongoose.model('Model', ModelnameSchema); // Exporting our model for use (usually in the
controller)
```

Once we have required Mongoose, we have to define a schema. MongoDB is a lazy database in that it doesn't enforce restrictions on records. It allows any type of data to be stored. It also allows new fields to be created on the fly without throwing any errors. Below is a table that will help you create an association between the **terminology** in **MongoDB VS a SQL database**:

MongoDB	SQL (MSSQL, MySQL, PostgreSQL)
Database	Database
Collection	Table
Document	Record
Attribute	Field

The Blog Model

Hopefully you understand everything so far and that and you're still with me. Let's add our logic into our **blog.js** model:

models/blog.js

```
// We will need our mongoose library
const mongoose = require(`mongoose`);

// Our schema
const BlogSchema = new mongoose.Schema({
  title: {
    type: String,
    required: true // we will require a title for our blog
  },
  content: {
    type: String,
    required: false
  },
  status: {
    type: String,
    enum: ['DRAFT', 'PUBLISHED'],
    default: 'DRAFT'
  }
}, {
  timestamps: true // we want timestamps to keep track of when our blogs were created
});

// Query Helper
// A helper that will return all blogs with the status of draft
BlogSchema.query.drafts = function () {
  return this.where({
    status: 'DRAFT'
  });
};

// A helper that will return all blogs with the status of published
BlogSchema.query.published = function () {
  return this.where({
    status: 'PUBLISHED'
  });
};

// Exporting our blog model
module.exports = mongoose.model('Blog', BlogSchema);
```

We only require 3 fields for our blog model (currently). We need a **title**, a field for **content**, and a way to define if our blog is **published** or still in **draft** mode. This will make it easier to determine if a general user should be able to see the blog or not. We define their datatypes as **string** for all 3 fields because that's what we'll be storing. We have other options including **boolean**, **number**, **array**, and **object**, not including custom data types created using additional schemas (which we'll look at in the next lesson).

Next we create 2 methods so we can easily get all of the blogs that are in **published** mode or **draft** mode. Really this is just function that wraps existing Mongoose model logic. If we have a common query we'll be performing a lot, it's helpful to use the built in **Mongoose query helpers** utility to build out those queries. This simplifies our access as we'll see in our blog controller in a bit.

BREAK

Create & Read Routes & Controller Actions

The Blogs Controller (stubbin' it)

We are now ready to hookup our controller. Our app will be failing up to this point if we attempt to access any of those routes we

created. Let's stub out the actions we'll need so our routes will function:

controllers/blogsController.js

```
const Blog = require('../models/blog');

exports.index = (req, res) => {}


exports.show = (req, res) => {}


exports.drafts = (req, res) => {}


exports.published = (req, res) => {}


exports.new = (req, res) => {}


exports.create = (req, res) => {}
```

Notice we're requiring our **blog model**. This will allow us to query the model for existing records or create/update/delete records. I've added the actions for **drafts** and **published** but we're missing the routes. Let's add those in now into our **routes/blogs.js** file:

routes/blogs.js

```
// Our router module
const router = require('express').Router();

// Our controller
const BlogsController = require('../controllers/blogsController');

// Our routes
router.get(`/`, BlogsController.index);
router.get(`/new`, BlogsController.new);
router.get(`/drafts`, BlogsController.drafts);
router.get(`/published`, BlogsController.published);
router.get(`/:id`, BlogsController.show);
router.post(`/`, BlogsController.create);

// We have to export our changes
module.exports = router;
```

Don't forget: It is important that the **/drafts** and the **/published** go **above** the **/:id** so they don't accidentally get interpreted as arguments.

Our Views

We will need to flesh out our views for the **index**, **show**, and **new** actions in our controller. I have included the logic for these below:

views/blogs/index.pug

```
extends ../layouts/main.pug
block content
  .container
    header
      h1= title

    div
      table.table.table-striped
        thead
          tr
            th Title
            th Status

        tbody
          each blog in blogs
            tr
              td
                a(href=`/blogs/${blog.id}`)= blog.title
              td= blog.status
```

Pug supports iteration allowing us to loop over an array and use its values. In the logic above we're iterating over **blogs** which is an array we'll be sending to our view from our blogs controller. Each blog record will have the attributes defined in our schema. We can access these as properties allowing us to view their values.

views/blogs/show.pug

```
extends ../layouts/main.pug
block content
  .container
    header
      h1= blog.title
      small= blog.status.toLowerCase()

    div
      = blog.content || "Your story is processing..."
```

The = tells **Pug** to evaluate the following as JavaScript and convert it to a string. Simply it will display whatever the result is in HTML. This is very powerful because (as you can see above) we can perform conditional logic and show specific things based on its evalution.

views/blogs/new.pug

```
extends ../layouts/main.pug
block content
  .container
    header
      h1 New Blog Post

    div
      form(action="/blogs", method="POST")
        .form-group
          label Title
          input.form-control(name="blog[title]", required)

        .form-group
          label Content
          textarea.form-control(name="blog[content]")

        .form-group
          label Status
          select.form-control(name="blog[status]", required)
            option(value="DRAFT") Draft
            option(value="PUBLISHED") Publish

        .form-group
          button.btn.btn-dark(type="submit") Submit
```

This is our first form. Notice our action is pointing to **/blogs** and our method is **POST**. This will resolve to the controller action **create**. When we submit the form, our form values will become an object that we will have access to within our controller. We can then write those values to our model in our MongoDB.

Body Parser

Body Parser is essential to translating our request bodies. It will convert our query parameters (localhost:4000?id=1234&name=shaun) and post bodies (form data) into JavaScript Object Notation, or JSON. This makes working with this data considerably easier. One nice thing about the body parser is we already have it and don't need to install it:

app.js (currently)

```
// Our dotenv
require('dotenv').config();

// Connecting to MongoDB cluster with Mongoose
const mongoose = require('mongoose');
mongoose.connect(process.env.DB_URI, {
  auth: {
    user: process.env.USERNAME,
    password: process.env.PASSWORD
  },
  useNewUrlParser: true
}).catch(err => console.error(`ERROR: ${err}`));

// Our imported libraries
const express = require('express');

// Assigning Express to an app contstant
const app = express();

// This maintains our home path
const path = require('path');

// Body parser which will make reading request bodies MUCH easier
const bodyParser = require('body-parser');
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({
  extended: true
}));

// Our Views
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
app.use('/css', express.static('assets/stylesheets'));
app.use('/js', express.static('assets/javascripts'));
app.use('/images', express.static('assets/images'));

// Our routes
const routes = require('./routes.js');
app.use('/', routes);

// Starting our server on port 4000
app.listen((process.env.PORT || 4000), () => console.log('Listening on 4000'));
```

First we require the body parser library and assign it to a variable. Next, we register the body parser with our application. This will tell Express to use the body parser to translate our request bodies. In addition we tell the app we want to translate using JSON. Last we tell Express to use urlencoded which will translate query strings into JSON as well, otherwise this would only work with POST bodies.

Controller Actions

New

We'll start with new. This will deliver the user our form view.

controllers/blogsController.js

```
exports.new = (req, res) => {
  res.render('blogs/new', {
    title: `New Blog Post`
  });
}
```

New doesn't need access to our **Blog** model (yet) because we're don't need any extra data. However, this does introduce a new bit of syntax: **locals**. The idea behind locals is they're a collection of variables scoped to the current view you're in. **Locals** are passed as a second argument to **render** as an object. Each key in the object becomes an accessible variable and the value of the property is the value of the variable.

Create

Next we'll create create. This way we can view an existing set of records that we'll create to test our form.

controllers/blogsController.js

```
exports.create = (req, res) => {
  Blog.create(req.body.blog)
    .then(() => {
      res.redirect('/blogs');
    })
    .catch(err => {
      console.error(`ERROR: ${err}`);
    });
}
```

Create doesn't have a view to render as its sole purpose is to create a new blog post. It does this by first accessing our **Blog** model, then calling create and passing an object with our data. **req.body.blog** is the data from our form in the **new** view. If you didn't notice already, the field names in the form all began with **blog** followed by the array syntax **[]** and a property name. The attribute looked like this: **name="blog[title]"**. When our form is sent, the **bodyparser** module will pick this up and translate it into JSON allowing us to work with it as an object. This object can be found under **req.body**. We have made it even easier by grouping our three needed inputs, **title**, **content**, and **status** under **blog**, so our object to write to our collection will be **req.body.blog**.

Another way we could write this would be like so:

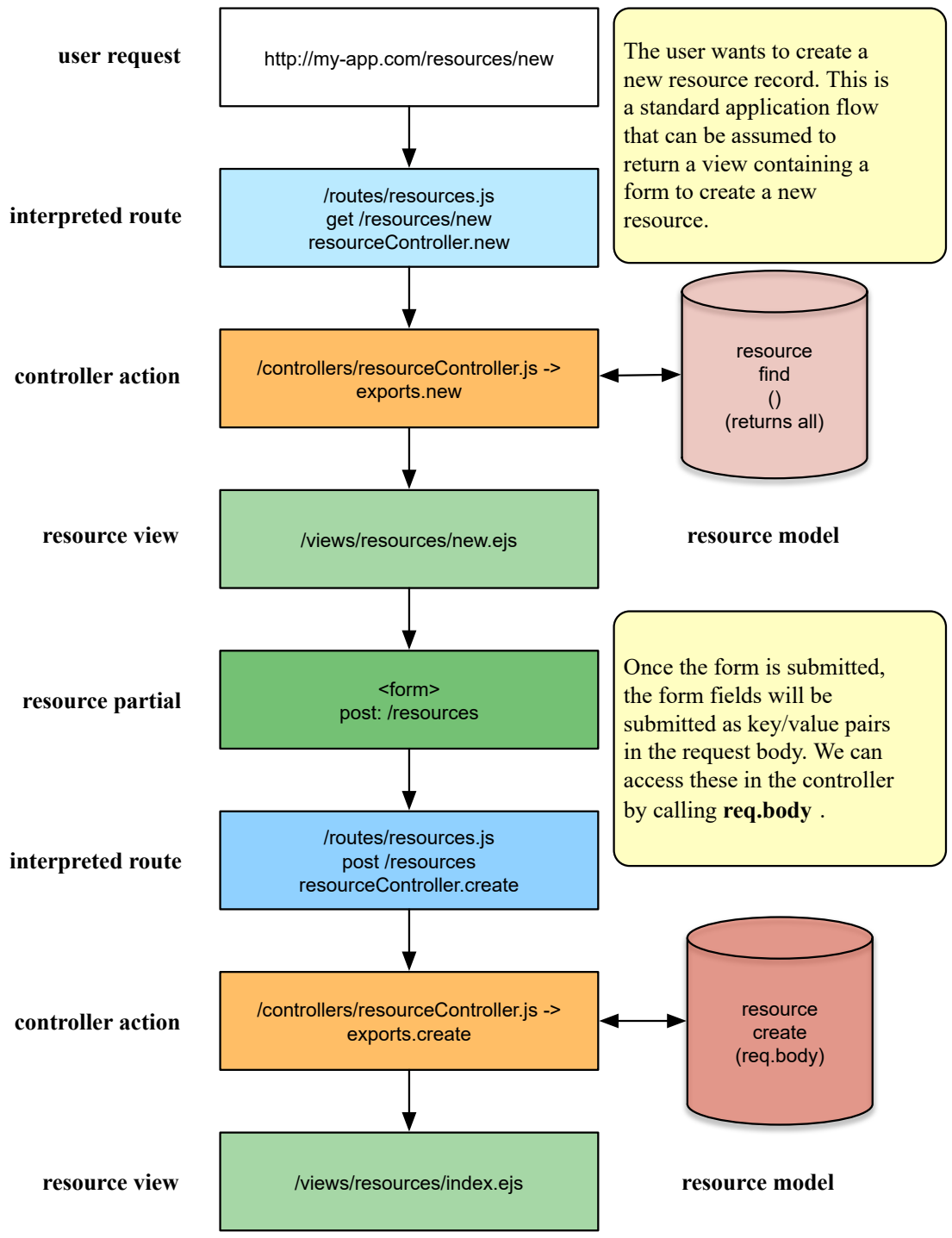
```
exports.create = (req, res) => {
  Blog.create({
    title: req.body.blog.title,
    content: req.body.blog.content,
    status: req.body.blog.status
  })
    .then(() => {
    ...
```

But because our object already will look like that, we can get away with just passing req.body.blog and avoid breaking it out. Sometimes we don't have that convenience.

Below is a flow diagram of how **new** and **create**will work.

1. The user makes a request for a new resource
2. Our router sends the request to **BlogsController.new**
3. The controller renders the **views/blogs/new** view
4. The user fills in the form and clicks submit
5. The router interprets the post and sends the request to **BlogsController.create**
6. The post is translated and passed to our **Blog model** to be **created**
7. If successful, we redirect the request to **/blogs/index**

8. The router send the request to **BlogsController.index**
9. ... (see index) ...



Index

This will show our existing records which will populate within our table view.

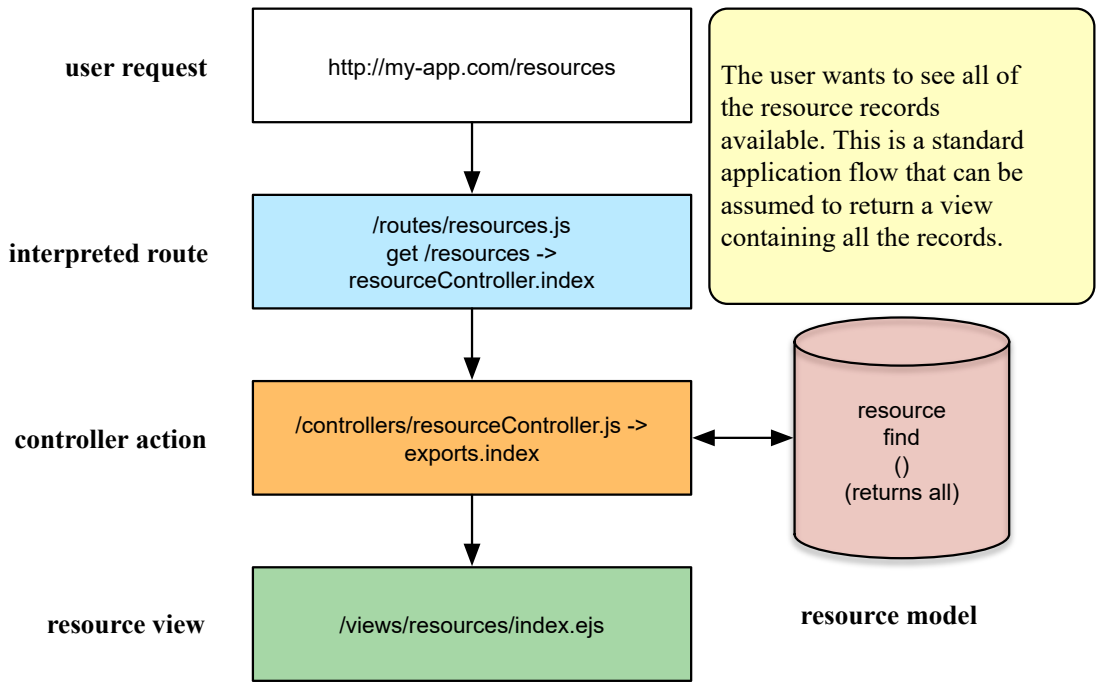
controllers/blogsController.js

```
exports.index = (req, res) => {
  Blog.find()
    .then(blogs => {
      res.render('blogs/index', {
        blogs: blogs,
        title: 'Archive'
      });
    })
    .catch(err => {
      console.error(`ERROR: ${err}`);
    });
}
```

Index is responsible for showing our user all the current blog posts. In our action we connect to **Blog** and perform a **find()**. find() in Mongoose is a query helper that will return all records that meet the condition. When we don't pass an argument to **find()** it will return all records. Once we have the records we want to pass them to our view. We're using **.then** and **.catch** to handle successes and errors. If **Blog** throws an error, our catch will execute its callback. If we get records and everything is successful **.then** will execute its callback which will render our **views/blogs/index** view and pass it a locals object. In the locals object we include our records to pass them to the view where we can render them.

Below is the application flow for **index**:

1. User makes a request to see all the resources
2. The router interprets the request and sends it to **BlogsController.index**
3. The controller action queries the database for all the blog documents
4. The controller then renders the **index** view and passes our blog documents to it



Show

This will show a singular record. We'll use the link in the table view to direct us to this action.

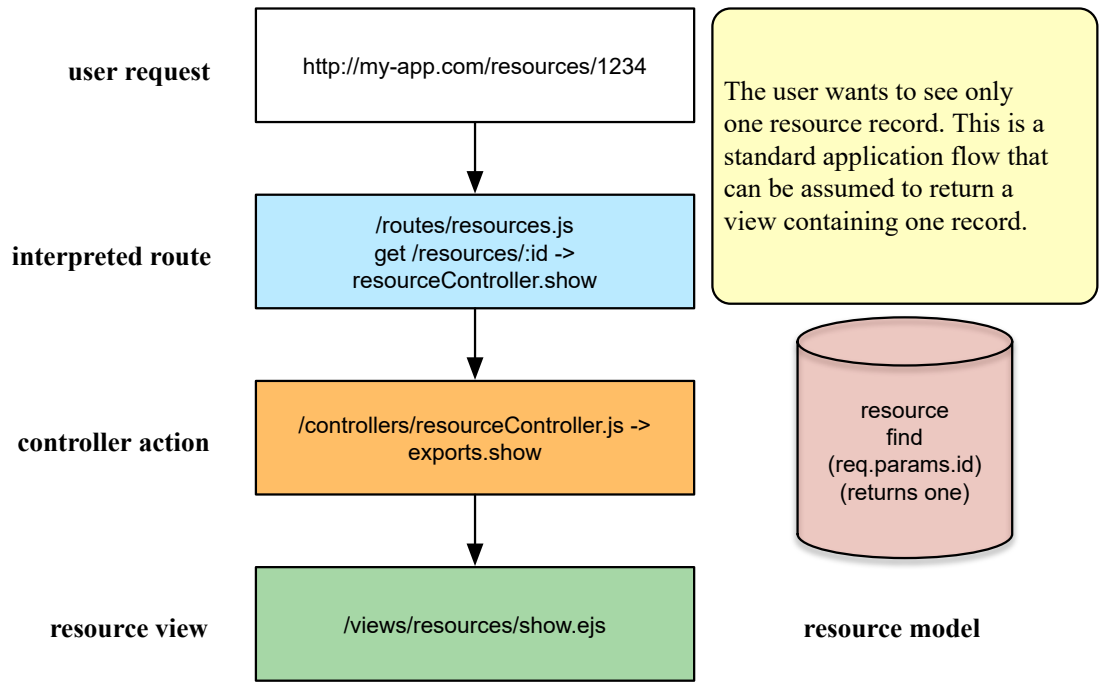
controllers/blogsController.js

```
exports.show = (req, res) => {
  Blog.findById(req.params.id)
    .then(blog => {
      res.render('blogs/show', {
        blog: blog,
        title: blog.title
      });
    })
    .catch(err => {
      console.error(`ERROR: ${err}`);
    });
}
```

Show works by having the router bind a variable to the query parameter passed by the user. **:id** was created to store the blog id the user is requesting. We can use that to pass it to our **Blog** model as a query. **.findById** is a prewritten query helper that will use the ID to find the requested dcoument and return a singular document (instead of an array of documents).

Below is an action flow for **show**:

1. The user requests to see a single blog post
2. The router interprets the request and translates the bound parameter **:id** as query string parameter
3. Our application translates that into a property on the params object
4. We use this to query **Blog** model for the document
5. We then render **show** and pass it our single blog post



Draft & Published

This will filter our blogs to either show only draft version or published versions. We'll use the same index view but send a different title and set of blog posts.

controllers/blogsController.js

```
exports.drafts = (req, res) => {
  Blog.find().drafts()
    .then(drafts => {
      res.render('blogs/index', {
        blogs: drafts,
        title: 'Drafts'
      });
    })
    .catch(err => {
      console.error(`ERROR: ${err}`);
    });
}


exports.published = (req, res) => {
  Blog.find().published()
    .then(published => {
      res.render('blogs/index', {
        blogs: published,
        title: 'Published'
      });
    })
    .catch(err => {
      console.error(`ERROR: ${err}`);
    });
}
```

Draft and **Published** we'll call their query helpers in order to find draft blogs or published blogs. This filtering will make it easy to show only one of the statuses at a time. Because **index**doesn't really care what the blogs collection we pass has in it, we can get away with using the same view for our filtered blogs.

Models & MongoDB Part 2

Create & Read Routes & Controller Actions

The Blogs Controller (stubbin' it)

We are now ready to hookup our controller. Our app will be failing up to this point if we attempt to access any of those routes we created. Let's stub out the actions we'll need so our routes will function:

controllers/blogsController.js

```
const Blog = require('../models/blog');

exports.index = (req, res) => {}


exports.show = (req, res) => {}


exports.drafts = (req, res) => {}


exports.published = (req, res) => {}


exports.new = (req, res) => {}


exports.create = (req, res) => {}
```

Notice we're requiring our **blog model**. This will allow us to query the model for existing records or create/update/delete records. I've added the actions for **drafts** and **published** but we're missing the routes. Let's add those in now into our **routes/blogs.js** file:

routes/blogs.js

```
// Our router module
const router = require('express').Router();

// Our controller
const BlogsController = require('../controllers/blogsController');

// Our routes
router.get(`/`, BlogsController.index);
router.get(`/new`, BlogsController.new);
router.get(`/drafts`, BlogsController.drafts);
router.get(`/published`, BlogsController.published);
router.get(`/:id`, BlogsController.show);
router.post(`/`, BlogsController.create);

// We have to export our changes
module.exports = router;
```

Don't forget: It is important that the **/drafts** and the **/published** go **above** the **/:id** so they don't accidentally get interpreted as arguments.

Our Views

We will need to flesh out our views for the **index**, **show**, and **new** actions in our controller. I have included the logic for these below:

views/blogs/index.pug

```
extends ../layouts/main.pug
block content
  .container
    header
      h1= title

    div
      table.table.table-striped
        thead
          tr
            th Title
            th Status

        tbody
          each blog in blogs
            tr
              td
                a(href=`/blogs/${blog.id}`)= blog.title
              td= blog.status
```

Pug supports iteration allowing us to loop over an array and use its values. In the logic above we're iterating over **blogs** which is an array we'll be sending to our view from our blogs controller. Each blog record will have the attributes defined in our schema. We can access these as properties allowing us to view their values.

views/blogs/show.pug

```
extends ../layouts/main.pug
block content
  .container
    header
      h1= blog.title
      small= blog.status.toLowerCase()

    div
      = blog.content || "Your story is processing..."
```

The = tells **Pug** to evaluate the following as JavaScript and convert it to a string. Simply it will display whatever the result is in HTML. This is very powerful because (as you can see above) we can perform conditional logic and show specific things based on its evalution.

views/blogs/new.pug

```
extends ../layouts/main.pug
block content
  .container
    header
      h1 New Blog Post

    div
      form(action="/blogs", method="POST")
        .form-group
          label Title
          input.form-control(name="blog[title]", required)

        .form-group
          label Content
          textarea.form-control(name="blog[content]")

        .form-group
          label Status
          select.form-control(name="blog[status]", required)
            option(value="DRAFT") Draft
            option(value="PUBLISHED") Publish

        .form-group
          button.btn.btn-dark(type="submit") Submit
```

This is our first form. Notice our action is pointing to **/blogs** and our method is **POST**. This will resolve to the controller action **create**. When we submit the form, our form values will become an object that we will have access to within our controller. We can then write those values to our model in our MongoDB.

Body Parser

Body Parser is essential to translating our request bodies. It will convert our query parameters (localhost:4000?id=1234&name=shaun) and post bodies (form data) into JavaScript Object Notation, or JSON. This makes working with this data considerably easier. One nice thing about the body parser is we already have it and don't need to install it:

app.js (currently)

```
// Our dotenv
require('dotenv').config();

// Connecting to MongoDB cluster with Mongoose
const mongoose = require('mongoose');
mongoose.connect(process.env.DB_URI, {
  auth: {
    user: process.env.USERNAME,
    password: process.env.PASSWORD
  },
  useNewUrlParser: true
}).catch(err => console.error(`ERROR: ${err}`));

// Our imported libraries
const express = require('express');

// Assigning Express to an app contstant
const app = express();

// This maintains our home path
const path = require('path');

// Body parser which will make reading request bodies MUCH easier
const bodyParser = require('body-parser');
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({
  extended: true
}));

// Our Views
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
app.use('/css', express.static('assets/stylesheets'));
app.use('/js', express.static('assets/javascripts'));
app.use('/images', express.static('assets/images'));

// Our routes
const routes = require('./routes.js');
app.use('/', routes);

// Starting our server on port 4000
app.listen((process.env.PORT || 4000), () => console.log('Listening on 4000'));
```

First we require the body parser library and assign it to a variable. Next, we register the body parser with our application. This will tell Express to use the body parser to translate our request bodies. In addition we tell the app we want to translate using JSON. Last we tell Express to use urlencoded which will translate query strings into JSON as well, otherwise this would only work with POST bodies.

Controller Actions

New

We'll start with new. This will deliver the user our form view.

controllers/blogsController.js

```
exports.new = (req, res) => {
  res.render('blogs/new', {
    title: `New Blog Post`
  });
}
```

New doesn't need access to our **Blog** model (yet) because we're don't need any extra data. However, this does introduce a new bit of syntax: **locals**. The idea behind locals is they're a collection of variables scoped to the current view you're in. **Locals** are passed as

a second argument to **render** as an object. Each key in the object becomes an accessible variable and the value of the property is the value of the variable.

Create

Next we'll create create. This way we can view an existing set of records that we'll create to test our form.

controllers/blogsController.js

```
exports.create = (req, res) => {
  Blog.create(req.body.blog)
    .then(() => {
      res.redirect('/blogs');
    })
    .catch(err => {
      console.error(`ERROR: ${err}`);
    });
}
```

Create doesn't have a view to render as its sole purpose is to create a new blog post. It does this by first accessing our **Blog** model, then calling create and passing an object with our data. **req.body.blog** is the data from our form in the **new** view. If you didn't notice already, the field names in the form all began with **blog** followed by the array syntax **[]** and a property name. The attribute looked like this: **name="blog[title]"**. When our form is sent, the **bodyparser** module will pick this up and translate it into JSON allowing us to work with it as an object. This object can be found under **req.body**. We have made it even easier by grouping our three needed inputs, **title**, **content**, and **status** under **blog**, so our object to write to our collection will be **req.body.blog**.

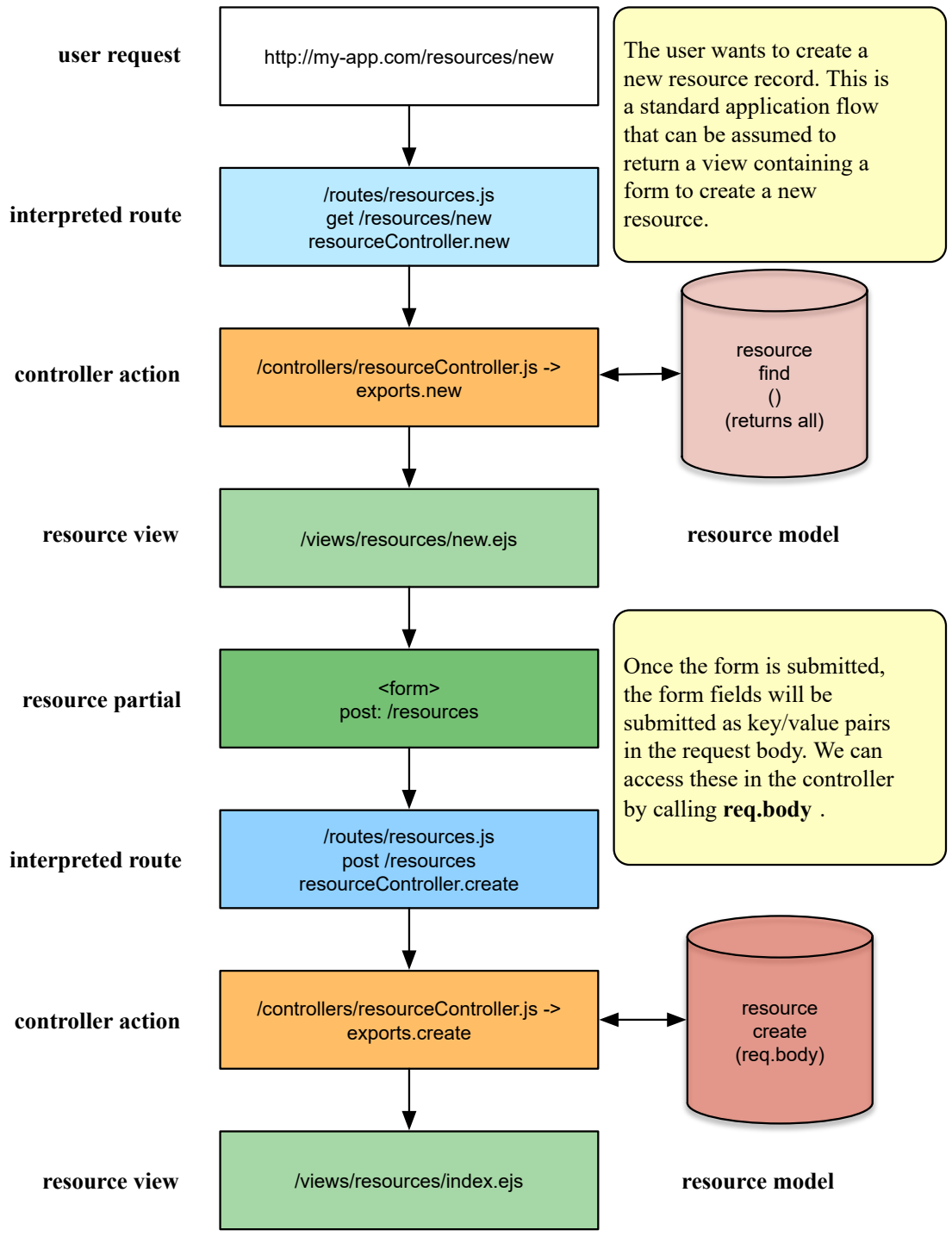
Another way we could write this would be like so:

```
exports.create = (req, res) => {
  Blog.create({
    title: req.body.blog.title,
    content: req.body.blog.content,
    status: req.body.blog.status
  })
    .then(() => {
    ...
```

But because our object already will look like that, we can get away with just passing req.body.blog and avoid breaking it out. Sometimes we don't have that convenience.

Below is a flow diagram of how **new** and **create**will work.

1. The user makes a request for a new resource
2. Our router sends the request to **BlogsController.new**
3. The controller renders the **views/blogs/new** view
4. The user fills in the form and clicks submit
5. The router interprets the post and sends the request to **BlogsController.create**
6. The post is translated and passed to our **Blog model** to be **created**
7. If successful, we redirect the request to **/blogs/index**
8. The router send the request to **BlogsController.index**
9. ... (see index) ...



Index

This will show our existing records which will populate within our table view.

controllers/blogsController.js

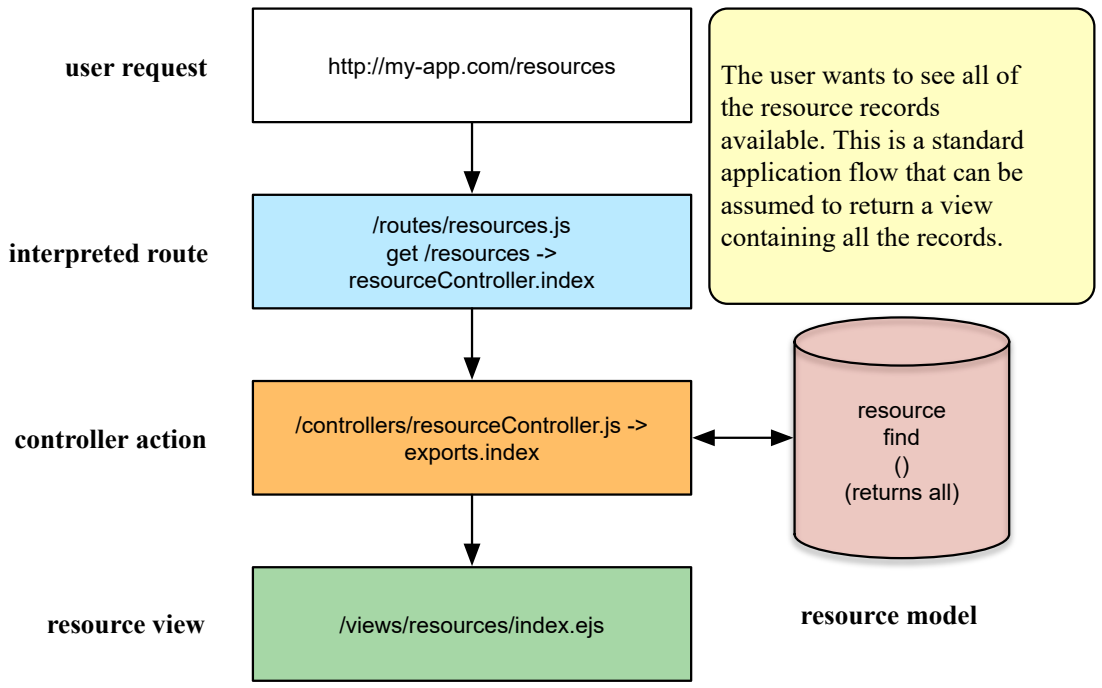
```
exports.index = (req, res) => {
  Blog.find()
    .then(blogs => {
      res.render('blogs/index', {
        blogs: blogs,
        title: 'Archive'
      });
    })
    .catch(err => {
      console.error(`ERROR: ${err}`);
    });
}
```

Index is responsible for showing our user all the current blog posts. In our action we connect to **Blog** and perform a **find()**. find() in Mongoose is a query helper that will return all records that meet the condition. When we don't pass an argument to **find()** it will return all records. Once we have the records we want to pass them to our view. We're using **.then** and **.catch** to handle successes and errors. If **Blog** throws an error, our catch will execute its callback. If we get records and everything is successful **.then** will execute its callback which will render our **views/blogs/index** view and pass it a locals object. In the locals object we include our records to pass them to the view where we can render them.

Below is the application flow for **index**:

1. User makes a request to see all the resources
2. The router interprets the request and sends it to **BlogsController.index**

3. The controller action queries the database for all the blog documents
4. The controller then renders the **index** view and passes our blog documents to it



Show

This will show a singular record. We'll use the link in the table view to direct us to this action.

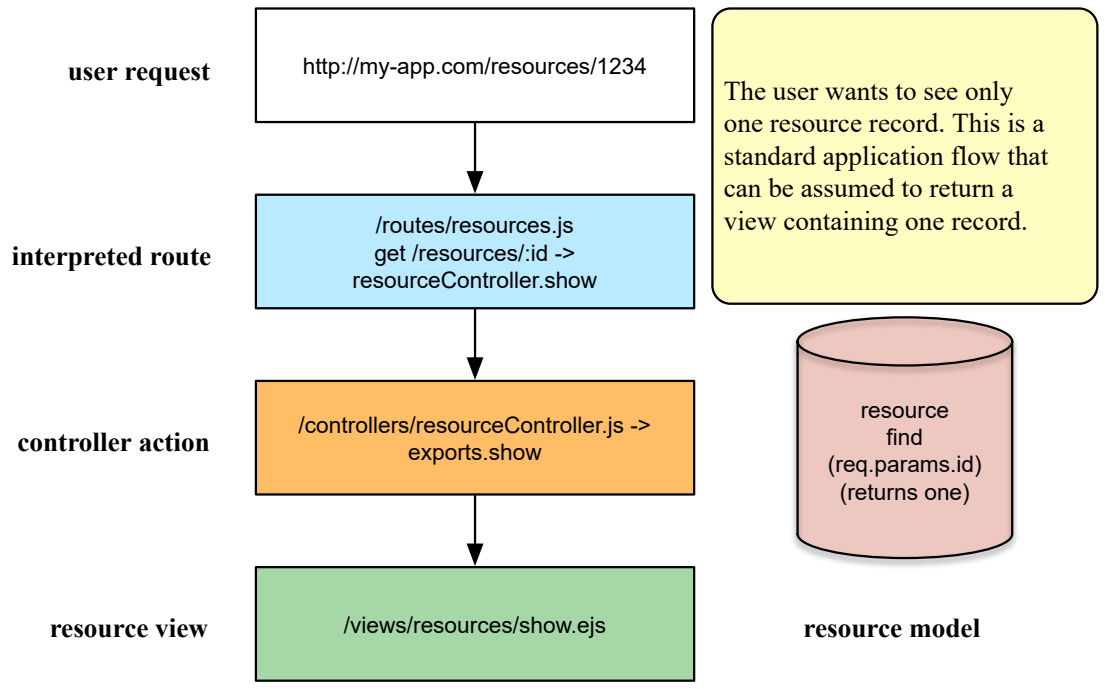
controllers/blogsController.js

```
exports.show = (req, res) => {
  Blog.findById(req.params.id)
    .then(blog => {
      res.render('blogs/show', {
        blog: blog,
        title: blog.title
      });
    })
    .catch(err => {
      console.error(`ERROR: ${err}`);
    });
}
```

Show works by having the router bind a variable to the query parameter passed by the user. **:id** was created to store the blog id the user is requesting. We can use that to pass it to our **Blog** model as a query. **.findById** is a prewritten query helper that will use the ID to find the requested dcoument and return a singular document (instead of an array of documents).

Below is an action flow for **show**:

1. The user requests to see a single blog post
2. The router interprets the request and translates the bound parameter **:id** as query string parameter
3. Our application translates that into a property on the params object
4. We use this to query **Blog** model for the document
5. We then render **show** and pass it our single blog post



Draft & Published

This will filter our blogs to either show only draft version or published versions. We'll use the same index view but send a different title and set of blog posts.

controllers/blogsController.js

```
exports.drafts = (req, res) => {
  Blog.find().drafts()
    .then(drafts => {
      res.render('blogs/index', {
        blogs: drafts,
        title: 'Drafts'
      });
    })
    .catch(err => {
      console.error(`ERROR: ${err}`);
    });
}


exports.published = (req, res) => {
  Blog.find().published()
    .then(published => {
      res.render('blogs/index', {
        blogs: published,
        title: 'Published'
      });
    })
    .catch(err => {
      console.error(`ERROR: ${err}`);
    });
}
```

Draft and **Published** we'll call their query helpers in order to find draft blogs or published blogs. This filtering will make it easy to show only one of the statuses at a time. Because **index**doesn't really care what the blogs collection we pass has in it, we can get away with using the same view for our filtered blogs.

BREAK

Editing, Updating, & Deleting a Resource

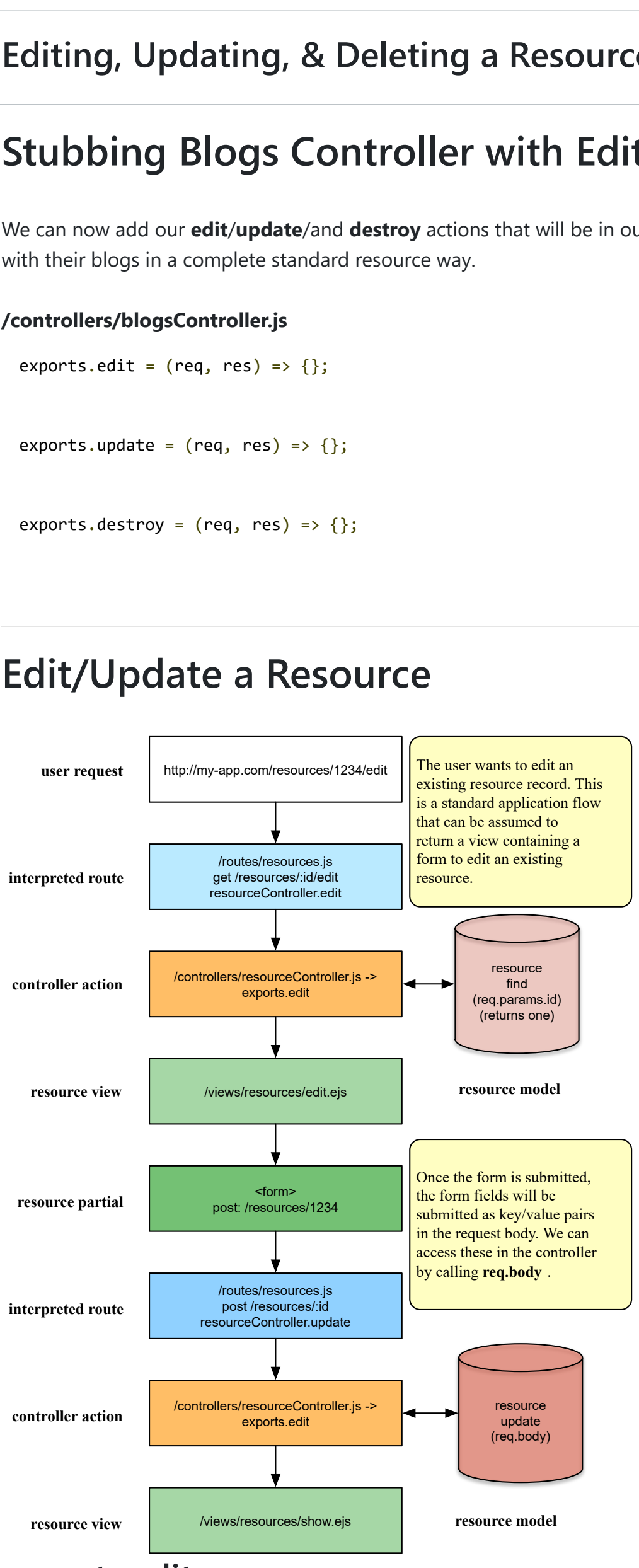
Stubbing Blogs Controller with Edit/Destroy

We can now add our **edit**/**update**/and **destroy** actions that will be in our blogs controller. These will allow our users to interact with their blogs in a complete standard resource way.

/controllers/blogsController.js

```
exports.edit = (req, res) => {};


exports.update = (req, res) => {};


exports.destroy = (req, res) => {};
```

Edit/Update a Resource



exports.edit

This method will deliver a view for the user to fill in a form. Because **new** is the exact same form, it makes sense to extract our form into a **partial** and then allow **new** and **edit** to **include** the form.

/views/blogs/form.pug

```
form(action=(blog ? `/blogs/update` : "/blogs"), method="POST")
  if blog
    input(type="hidden", name="id", value=blog.id)

  .form-group
    label Title
    input.form-control(name="blog[title]", required, value=(blog ? blog.title : null))

  .form-group
    label Content
    textarea.form-control(name="blog[content]")
      = blog ? blog.title : null

  .form-group
    label Status
    select.form-control(name="blog[status]", required)
      each status in ['DRAFT', 'PUBLISHED']
        option(value=status, selected=(blog && blog.status === status ? true : false))= status.toLowerCase()

  .form-group
    button.btn.btn-dark(type="submit") Submit
```

Here we're taking advantage of several awesome aspects of the **Pug template engine:**

- the form action will change based on being **new** or **edit**, so we use a **ternary** to define the **action path**
- we will want to pass the **blog.id** to our **controller action**, **update** so it can do a lookup and update the appropriate document. We only need to this field if we're **editing** our blog, so we can use **Pugscondition syntax** (the **if** statement) to control its visibility
- if our form is for **edit**ing then we want to prepopulate the value. We can do this by using a **ternary** to show the **blog.title** if our **local**, **blog**, is available (**REMEMBER:** that locals are variables **local** to the current view. They are passed by the controller)
- because our logic for the status options is the same, we can take advantage of **Pug iterators** and **DRY**up our application a bit
    - we're iterating through the array of statuses and assigning it to a **status** variable
    - now we can use a ternary to choose a selected value based on the currently selected value

Now that we've abstracted the form, we can modify **new.pug** to include the view, then create our **edit.pug** view with the same logic:

/views/blogs/new.pug

```
extends ../layouts/main.pug
block content
  .container
    header
      h1 New Blog Post

    div
      include ./form.pug
```

/views/blogs/edit.pug

```
extends ../layouts/main.pug
block content
  .container
    header
      h1= `Edit ${blog.title}`

    div
      include ./form.pug
```

We can take advantage of having the **blog** document and personalize our title accordingly using string interpolation.

Lastly, we can add the logic to our **exports.edit** action in the **blogsController:**
/controllers/blogs.js

```
exports.edit = (req, res) => {
  Blog.findById(req.params.id)
    .then(blog => {
      res.render('blogs/edit', {
        title: `Edit ${blog.title}`,
        blog: blog
      })
    })
    .catch(err => {
      console.error(`ERROR: ${err}`);
    });
}
```

First we find our blog by our ID that we get from our parameters (passed by our bound parameter from our router). If we're successful, we'll render our view and pass our locals, otherwise we'll throw an error.

exports.update

When we send the **blog post edit form** we will need to access our **Blog model** and modify its details. One important thing to note is **Mongoose** doesn't automatically validate updated documents. We will need to tell Mongoose we want to run our validators when we attempt to update our blog. We do this by passing in an extra argument.

The method we'll be using to do the update is called **model.updateOne()**. It takes 3 arguments (1 is optional):

```
model.updateOne({filter: value}, {set1: value, set2: value}, {option: value});
```

Filter is what we want to search by. For example, **{_id: jas9098asdfkjlk08}**. Here we want any record that contains the id of jas9098asdfkjlk08.
Set is our second argument. Set is the attribute we want to modify and the value we want to change it to.
Option is the third (optional) argument that tells Mongoose what options we want to run our query with. For example, we can tell Mongoose to **upsert** which will cause Mongoose to insert a new record if it cannot find an existing one. **runValidators** will tell Mongoose that we wish to run our update by our validation before we attempt to write.

/controllers/blogsController.js

```
exports.update = (req, res) => {
  Blog.updateOne({
      _id: req.body.id
    }, req.body.blog, {
      runValidators: true
    })
    .then(() => {
      res.redirect('/blogs');
    })
    .catch(err => {
      console.error(`ERROR: ${err}`);
    });
};
```

Our first argument is the _id we want to find. **updateOne** seems to enforce the id column by its MongoDB version, **_id**. The second argument is our form body. In the form, we've filtered our fields to be under **blog**. This will allow us to easily only insert the relevant data and not the **id** property accidentally. Lastly we tell Mongoose we want to run our update against our validation checks. If it's rejected, we'll catch the error and console log it. Otherwise we'll redirect to our index page.

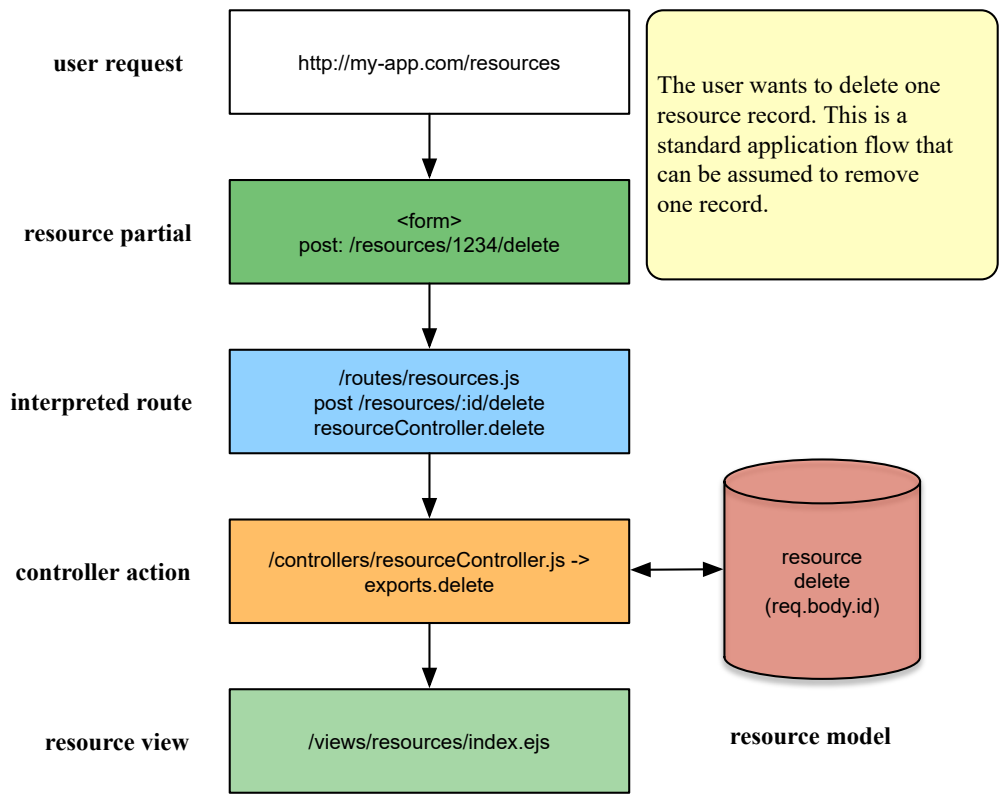
Before we go, we should update our **blogs/index.pug** view so it gives us a way to access these blogs for editing. We can just add a simple link to achieve what we need:

/views/blogs/index.pug

```
extends ../layouts/main.pug
block content
  .container
    header
      h1= title

    div
      table.table.table-striped
        thead
          tr
            th Title
            th Status
            th

        tbody
          each blog in blogs
            tr
              td
                a(href=`/blogs/${blog.id}`)= blog.title
              td= blog.status
              td
                a(href=`/blogs/${blog.id}/edit`) edit
```

Notice we've added a new row in our thead and a new row in our tbody. We'll use string interpolation to format the path to what our router will require.

Delete a Resource



exports.destroy

Destroying a document requires us to find a document. We provide our action the id so we'll use that to find the document. The method we'll use is **model.deleteOne()**.

```
exports.destroy = (req, res) => {
  Blog.deleteOne({
      _id: req.body.id
    })
    .then(() => {
      res.redirect('/blogs');
    })
    .catch(err => {
      console.error(`ERROR: ${err}`);
    });
};
```

First we find the document we want to destroy by the **id**. If the document was destroyed successfully, we will then redirect to our index page. Otherwise, we'll catch the error and console log it out.

Destroying a document is an obvious destructive action. This means it should be a **post** request to signify what we want to do is destructive. This means that we will need to submit a form instead of navigating to a link. Let's modify our **blogs/index.pug** to reflect the new logic:

/views/blogs/index.pug

```
extends ../layouts/main.pug
block content
  .container
    header
      h1= title

    div
      table.table.table-striped
        thead
          tr
            th Title
            th Status
            th

        tbody
          each blog in blogs
            tr
              td
                a(href=`/blogs/${blog.id}`)= blog.title
              td= blog.status
              td
                a(href=`/blogs/${blog.id}/edit`) edit
                = ` | `
                form.destroy(action=`/blogs/${blog.id}/destroy`, method="POST")
                  input(type="hidden", name="blog[id]", value=`${blog.id}`)
                  input.destroy(type="submit", value="destroy")
```

We'll give our form a class called **destroy**. We'll use this to style our form's submit button to look like a standard link. Copy the below CSS styles and add them to **assets/stylesheets/styles.css:**

/assets/stylesheets/styles.css

```
.destroy {
  display: inline;
}

.destroy>input[type="submit"] {
  margin: 0;
  padding: 0;
  background: none;
  border: 0;
  color: #007bff;
}

.destroy:hover>input[type="submit"] {
  text-decoration: underline;
}

.container {
  margin-top: 60px;
}
```

The first 3 rules will modify our form's submit button to look and behave like a link. The last rule is to correct an issue caused by the navigation.

The New Routes

Now that we have 3 new controller actions, we will need 3 new routes in our blog router.

/routes/blogs.js

```
// Our router module
const router = require('express').Router();

// Our controller
const BlogsController = require('../controllers/blogsController');

// Our routes
router.get(`/`, BlogsController.index);
router.get(`/new`, BlogsController.new);
router.get(`/drafts`, BlogsController.drafts);
router.get(`/published`, BlogsController.published);
router.get(`/:id`, BlogsController.show);
router.get(`/:id/edit`, BlogsController.edit);
router.post(`/`, BlogsController.create);
router.post(`/update`, BlogsController.update);
router.post(`/destroy`, BlogsController.destroy);

// We have to export our changes
module.exports = router;
```

Edit will be a view, and therefore only consist of reading data. Because it isn't modifying data in any way, we will use the **GET method**. **Update** and **Destroy** are manipulators, and therefore will require us to use the **POST method** to router our request. Notice in **edit** we're using **bound parameters** to allow us to easily lookup our record. In the **update** and **destroy**actions we are not. This is because we're passing the ids through hidden fields instead.

BREAK

Flashing

Adding Flash Messaging

Flash messaging is a methodology for providing a better user experience. It allows us to take complex errors and successes and put them into a human readable format for the user. This helps the user to better understand what is happening with their application progress.

Flash messaging is a key component in many application frameworks. Getting it working requires a basic understanding of sessions and cookies, as there isn't always a guarantee we're coming to the next page directly from the controller. Because the browser is stateless, we need someway to keep track of our flash message between state changes.

terminal
We will need to install some libraries in order to build some flash messages. We will need the flash module, sessions, and cookie parser. Sessions allow us to communicate between 2 pages in our application. You have to remember that the browser is stateless so in order to maintain data between multiple pages we have to use a storage solution like sessions. In Express we use Cookie Parser to allow us to easily store cookies. Sessions are used to create a session identification marker that is unique to us. These will allow us to pass flash messages to from page to page with ease.

```
npm install express-session
npm install cookie-parser
npm install connect-flash
```

Registering our Components

We will need our cookie parser and session manager in order to move flash messages between states in the client. In order to make sessions work we need to pass a cookie an expiry time period. This will prevent the cookies and sessions from overstaying their welcome and force them to clear after a period of time.

/app.js

```
...

// Assigning Express to an app contstant
const app = express();

// Adding cookie and session support to our application
const cookieParser = require('cookie-parser');
const session = require('express-session');
const flash = require('connect-flash');
app.use(cookieParser());
app.use(session({
  secret: (process.env.secret || 'boorakacha'),
  cookie: {
    maxAge: 10800000
  },
  resave: true,
  saveUninitialized: true
}));
app.use(flash());
app.use((req, res, next) => {
  res.locals.flash = res.locals.flash || {};
  res.locals.flash.success = req.flash('success') || null;
  res.locals.flash.error = req.flash('error') || null;

  next();
});

...
```

Now we can add the flash messaging code to our controller.

/blogs/blogsController.js

```
...
exports.create = (req, res) => {
  Blog.create(req.body.blog)
    .then(() => {
      req.flash('success', 'New blog was created successfully.');
      res.redirect('/blogs');
    })
    .catch(err => {
      req.flash('error', `ERROR: ${err}`);
      res.redirect('/blogs/new');
    });
};

...

exports.update = (req, res) => {
  Blog.updateOne({
      _id: req.body.id
    }, req.body.blog, {
      runValidators: true
    })
    .then(() => {
      req.flash('success', 'The blog was updated successfully.');
      res.redirect('/blogs');
    })
    .catch(err => {
      req.flash('error', `ERROR: ${err}`);
      res.redirect(`/blogs/${req.body.id}/edit`);
    });
};


exports.destroy = (req, res) => {
  Blog.deleteOne({
      _id: req.body.id
    })
    .then(() => {
      req.flash('success', 'The blog was deleted successfully.');
      res.redirect('/blogs');
    })
    .catch(err => {
      req.flash('error', `ERROR: ${err}`);
      res.redirect(`/blogs`);
    });
};
```

Lastly we'll add a view to pickup our flash messages. It makes more sense to put this into its own partial for now.

/views/partials/flash.pug

```
if (flash)
  .flash
    if (flash.success.length)
      .alert.alert-success #{flash.success.join(`\n`)}
    if (flash.error.length)
      .alert.alert-danger #{flash.error.join(`\n`)}
```

One more step is to include the partial in our layout:

/views/layouts/main.pug

```
...

  body
    include ../partials/main-nav.pug
    include ../partials/flash.pug

    block content

...
```

Authentication & Association

Adding Authors

Simple Author Model

Currently we have no way to associate a blog post to a specific person, which means the site owner will own all the blog posts. It'd be nice to open our platform up to other authors. We may want to be more restrictive, but for now, this will do.

First we will need a new **Author** model that will allow us to add a new author. The **schema** for this is quite simple, even more so than the **blog** model we created in week 4:

/models/author.js

```
// We will need our mongoose library
const mongoose = require(`mongoose`);

// Our schema
const AuthorSchema = new mongoose.Schema({
  firstName: {
    type: String,
    required: true
  },
  lastName: {
    type: String,
    required: true
  },
  email: {
    type: String,
    required: true
  }
}, {
  timestamps: true
});

// Exporting our blog model
module.exports = mongoose.model('Author', AuthorSchema);
```

We will add considerably more complexity to this model later when we implement authentication. For now our fields are simple **String** fields that are all required.

Author Controller

The **AuthorsController** is very simple as it will only have 2 actions (currently). We should eventually implement the ability to edit an author or perhaps have an administrative level that allows us to manage all of the blog authors, but for now we'll keep it to just being able to register a new author:

/controllers/authorsController.js

```
const Author = require('../models/author');


exports.new = (req, res) => {
  res.render('authors/new', {
    title: `New Author`
  });
};


exports.create = (req, res) => {
  Author.create(req.body.author)
    .then(() => {
      req.flash('success', 'Your are now registered.');
      res.redirect('/login');
    })
    .catch(err => {
      req.flash('error', `ERROR: ${err}`);
      res.redirect('/authors/new');
    });
};
```

Like in the **BlogsController** we first require our **model**. This allows us to interact with our author resource and create a new one. The create method is the same, and, again, we're taking advantage of namespacing our author in our form (which you will see later).

Author Routes

Because we have such a simple controller, our routes are obviously quite simple as well:

/routes/authors.js

```
// Our router module
const router = require('express').Router();

// Our controller
const AuthorsController = require('../controllers/authorsController');

// Our routes
router.get(`/new`, AuthorsController.new);
router.post('/', AuthorsController.create);

// We have to export our changes
module.exports = router;
```

We will also need to register the new **authors routes** with our application:

/routes.js

```
...

// Importing the pageRoutes
...
const authorRoutes = require('./routes/authors');

// Registering our pageRoutes
...
app.use('/authors', authorRoutes);

...
```

Author View

We'll still follow the same view pattern we used for our **Blogs** by keeping our form to its own partial:

/views/authors/form.pug

```
form(action=(blog ? `/authors/update` : "/authors"), method="POST")
  if author
    input(type="hidden", name="id", value=author.id)

  .form-group
    label First Name
    input.form-control(name="author[firstName]", required, value=(author ? author.firstName : null))

  .form-group
    label Last Name
    input.form-control(name="author[lastName]", required, value=(author ? author.lastName : null))

  .form-group
    label Email
    input.form-control(name="author[email]", required, value=(author ? author.email : null))

  .form-group
    label Password
    input.form-control(name="author[password]", type="password", required, value=(author ? author.password : null))

  .form-group
    label Password Confirmation
    input.form-control(name="author[passwordConfirmation]", required, type="password", value=(author ?
author.passwordConfirmation : null))

  .form-group
    button.btn.btn-dark(type="submit") Submit
```

As you can see the logic is identical to the **blogs form.pug** view. Once we implement an edit action, these values will populate with our values.

Lastly we'll create the **new** view which will consume our form:

/views/authors/new.pug

```
extends ../layouts/main.pug
block content
  .container
    header
      h1 New Author

    div
      include ./form.pug
```


BREAK

Passwords and Authentication Helpers

BCrypt

https://www.npmjs.com/package/bcrypt

BCrypt is a popular library available in several languages. Essentially it lets you create a unidirectional hash from a plain string. Because it's unidirectional it is a lot harder to hack as the hacker must try pushing passwords against it repeatedly in order to crack it. Not impossible; just difficult.

To use BCrypt we'll need to install the node module:
npm install bcrypt

Next we'll add BCrypt to our **Author model**, provide it a salt factor, and add a password attribute to our schema:

/models/author.js

```
// Our schema
const AuthorSchema = new mongoose.Schema({
const bcrypt = require('bcrypt');
const SALT_WORK_FACTOR = 10;
  ...
  email: {
    type: String,
    required: true
  },
  password: {
    type: String,
    required: true
  }
  ...
```

The **SALT_WORK_FACTOR** will create an encrypted salt key that our password hash will utilize to generate a hashed password. The more factors your use, the more difficult it will be to decode for a hacker. 10 is the default that NodeJS uses, so we'll use the same with BCrypt.

Authentication Helpers

Virtual Attributes (Properties)

Before we build our authentication helpers we will need a way to confirm our user's password. Many systems have abandoned this concept and now provide a **show** button that will allow the user to see the password they've chosen. I'm of the mindset that that is very cavalier and leads to people seeing your new password over your shoulder. The password confirmation principle still feels the most secure and the optimal user experience.

In order to have a password confirmation check, we will need to use a **virtual attribute**. Virtual attributes allow us to have a field that can be referenced and used, but won't affect or be written to the MongoDB document. This means we can compare our password to the password confirmation attribute, but not write the password confirmation to our database by accident:

/models/author.js

```
// Our schema
...

// A virtual getter setter for passwordConfirmation
AuthorSchema.virtual('passwordConfirmation')
  .get(() => this.passwordConfirmation)
  .set((value) => this.passwordConfirmation = value);
```

The **model.virtual()** method will create a **virtual attribute** on our model, **Author** that we can then access. **.get()** creates a **getter**

method that will execute when we attempt to access the passwordConfirmation attribute on our model. The **.set()** creates a **setter** method that will set the value of our passwordConfirmation. This will, by default, set the attribute to the value we pass in our query **create**.

Helper Actions

When we created the **Draft** and **Published** actions in our **Blogs** model, we had to use **query helpers**. In addition to query helpers in Mongoose, we also have life cycle hooks. A life cycle hook will allow us to inject an operation at a particular point in a query's life cycle. The **.pre()** method will execute a callback at whatever point you choose. Below we're executing a callback to **hash the password before we save** it to Mongo:

/models/author.js

```
// A virtual getter setter for passwordConfirmation
...

// Our presave operations
// This will hash our password before we save
AuthorSchema.pre('save', function (next) {
  const author = this;
  if (!author.isModified('password')) return next;
  if (author.password !== author.passwordConfirmation) throw new Error('Your password does not match your password
confirmation');

  bcrypt.genSalt(SALT_WORK_FACTOR, (err, salt) => {
    if (err) return next(err);

    bcrypt.hash(author.password, salt, (err, hash) => {
      if (err) return next(err);

      author.password = hash;
      next();
    });
  });
});
```

When we create a new author document, the **.pre()** will run before validation and before writing the document to the Mongo database. We have to use a standard function callback over an arrow function as we need the associated context of the model to be available in the object. Because of this, we'll need to assign **this** to **author** to maintain owner context.

In our callback body we check to see if the **password** field has been modified. If it has, we skip past everything by calling **next()**. Otherwise we validate that the **password** and **passwordConfirmation** attributes match, otherwise we throw an error that will be picked up by whatever controller action invoked our pre save operation.

Once everything is validated, we generate a salt key then hash our password. Once we're finished, we call **next()** telling the Mongoose we're finished and letting it continue its life cycle.

Authenticate Helper

Once we have a **hashed password** we will need to be able to check plain text passwords against it. We can easily do this by using BCrypt's **.compare()** method. It takes the plain text password, the current hashed password, and a callback to function. The callback will potentially return an error or a boolean value stating whether the password matched. The error is strictly for syntax or logic errors. **isMatch** will contain our boolean. We will need to check this to see if it's true or false and return back the match through the callback. This is fairly complex so it's nice to have it contained in a simple helper.

```
...
// This will allow us to compare our password to plain text
AuthorSchema.methods.authenticate = function (plainPassword, callback) {
  // plain text, hash, callback
  bcrypt.compare(plainPassword, this.password, (err, isMatch) => {
    if (err) return callback(err);
    callback(null, isMatch);
  });
};

// Exporting our blog model
module.exports = mongoose.model('Author', AuthorSchema);
```


Sessions

What are Sessions?

A **session** is a managed period of time keyed to a client interacting with a server. Many servers will generate a session and give it a unique key and assign it the client that is accessing the server. Every client gets their own session. Sessions are the holy grail for hackers as they often contain very sensitive information or allow a user access that is otherwise restricted. Man-in-the-middle attacks make an attempt to possess a session so they can represent a client and their current role and credential status.

Often a server will generate a **session object**. A session object is like any object with properties and sometimes methods. This object will contain information about the session including information collected about the client such as IP address or client technology. Often applications will add their own content to a session object as it's a great way to maintain information between request states. For example, once a user has signed in a token will usually be written to a session property that can be used to validate the user is logged in. This eliminates the need to have the user login upon every request.

Routes

Most frameworks will keep **sessions** separated from **users** and manage them as their own resource. This is often because you may have different strategies depending on what information you want to share with an authenticated client. For example, a user may not login the same way as an API would. Having a separate **session resource** will permit adding new authentication strategies.

/routes/sessions.js

```
// Our router module
const router = require('express').Router();

// Our controller
const SessionsController = require('../controllers/sessionsController');

// Our routes
router.get(`/login`, SessionsController.login);
router.post(`/authenticate`, SessionsController.authenticate);
router.post(`/logout`, SessionsController.logout);

// We have to export our changes
module.exports = router;
```

We're only adding 3 routes to our **session router**. **SessionsController.login()** will render a view that contains a simple login form. Once the user has entered their credentials and clicked submit, then the **post request** will be sent to **SessionsController.authenticate()**. This will be the action where we verify our user and create a new sessions for them. When the user is finished, they can logout by sending a post request to **/logout**. This will call **SessionsController.logout** where we will

destroy the credential in our session, essentially deauthorizing them.

Links

In order to create new authors (or registered users) we'll need some links for the actions they need to be able to do. For now, all our links will be public:

/views/partials/main-nav.pug

```
nav.navbar.fixed-top.navbar-expand-lg.navbar-light.bg-light
  a.navbar-brand(href='/') The Blog
  button.navbar-toggler(type='button' data-toggle='collapse' data-target='#navbarSupportedContent' aria-
controls='navbarSupportedContent' aria-expanded='false' aria-label='Toggle navigation')
    span.navbar-toggler-icon

  #navbarSupportedContent.collapse.navbar-collapse
    ul.navbar-nav.mr-auto
      li.nav-item
        a.nav-link(href='/') Home
      li.nav-item
        a.nav-link(href='/about') About
        li.nav-item.dropdown
          a.nav-link.dropdown-toggle(href='#', data-toggle='dropdown', role='button', aria-haspopup='true', aria-
expanded='false') Blogs
          .dropdown-menu
            a.dropdown-item(href='/blogs') Blogs
            a.dropdown-item(href='/blogs/new') New Blog
            a.dropdown-item(href='/blogs/published') Published
            a.dropdown-item(href='/blogs/drafts') Drafts
        li.nav-item
          a.nav-link(href='/authors/new') Register
        li.nav-item
          a.nav-link(href='/login') Login
        li.nav-item
          form.nav-destroy(action=`/logout`, method="POST")
            input.destroy(type="submit", value="Logout")
      li.nav-item
        a.nav-link(href='/contact') Contact
```

The first 2 links, **Login** and **Register**, will point to simple view actions. **Register** will point to a new author and **Login** will point to a login form for the user to sign in. The last link, **Logout**, will require a form as it must be a post request that will execute the **destroy action** in the SessionsController.

Login

The **login action** will simply render the **login** view.

/contollers/sessionsController.js

```
const Author = require('../models/author');

exports.login = (req, res) => {
  res.render('sessions/login', {
    title: 'Login'
  });
};
```

Our view will be a simple form requesting the user to enter an email and a password. We will use the email to look up the user and

the **authenticate** helper to compare the password. Our action will obviously point to **/authenticate**, the route that will send the post request to **SessionsController.authenticate**.

/views/sessions/login.pug

```
extends ../layouts/main.pug
block content
  .container
    header
      h1 Login

    div
      form(action="/authenticate", method = "POST")
        .form-group
          label Email
          input.form-control(name="email", required)

        .form-group
          label Password
          input.form-control(name="password", type="password", required)

        .form-group
          button.btn.btn-dark(type="submit") Submit
```

Authenticate

When the user submits the form, the request will be sent to **SessionsController.authenticate**. Our first step will be to find the referenced author. **model.findOne()** takes an object as an argument that has keys and values used to do the lookup. This is similar to a **where clause** passed to a **SQL** statement. In our code below we're looking for a **single document** that contains an **email** with the passed form value. Once we have the author, we need to authenticate the password. We use **.authenticate** (our helper) to verify the user. It takes a callback that will either return an error or a boolean. We'll capture those in our callback parameters **err** and **isMatch**. If we have an error we'll throw the error. Keep in mind that the error would be a syntactical error or and implementation error, not an error caused by incorrect credentials.

If the credentials match we'll direct them to the blogs, otherwise we'll send back an error and redirect them to the login page for them to try again. Notice that we also create a new property (**userId**) on the **session**. **userId** will provide us a way of checking to see if our user is signed in or not. This way of authenticating isn't exactly the most secure but it is definitely the simplest. There are better way for implementing authentication such as **Passport**, a multi-strategy authentication implementation. However, to show the process of authentication in a granular way we're rolling our own implementation.

/controllers/sessionsController.js

```
exports.authenticate = (req, res) => {
  Author.findOne({
      email: req.body.email
    })
    .then(author => {
      author.authenticate(req.body.password, (err, isMatch) => {
        if (err) throw new Error(err);

        if (isMatch) {
          req.session.userId = author.id;

          req.flash('success', 'You are logged in.');
          res.redirect('/blogs');
        } else {
          req.flash('error', `ERROR: Your credentials do not match.`);
          res.redirect('/login');
        }
      });
    })
    .catch(err => {
      req.flash('error', `ERROR: ${err}`);
      res.redirect('/login');
    });
};
```

Logout

In order to logout, we simply have to get rid of the **userId** property we set. This will prevent the application from identifying the user as logged in.

/controllers/sessionsController.js

```
exports.logout = (req, res) => {
  req.session.userId = null;
  req.flash('success', 'You are logged out');
  res.redirect('/');
};
```

App Level Authentication Helpers

To make our lives **MUCH** easier we'll implement some **app level helpers** to verify if a user is logged in or not. Our function **isAuthenticated** will simply check that there's a session and a **session.userId** and return a boolean value. We'll create a new middleware and register it with our application. The first part will assign a function to **req.isAuthenticated**. That function will call **isAuthenticated**, pass in the request, and handle a flash message and an automatic redirect.

The second part will assign the boolean response to **res.locals.isAuthenticated**. This will allow us to check if the user is authenticated in our views by simply using the local **isAuthenticated**. Once we're done assigning functions and values, we need to call **next()** which will pass the **req**uest and **res**ponse on to the next middleware. If we forget to do that, the application would hang here.

/app.js

```
// Adding cookie and session support to our application
...
// Our authentication helper
const isAuthenticated = (req) => {
  return req.session && req.session.userId;
};
app.use((req, res, next) => {
  req.isAuthenticated = () => {
    if (!isAuthenticated(req)) {
      req.flash('error', `You are not permitted to do this action.`);
      res.redirect('/');
    }
  }

  res.locals.isAuthenticated = isAuthenticated(req);
  next();
});
// End of our authentication helper
...
```

Now that we have our helpers, we can add some authenticated level logic into our menu:

/views/partials/main-nav.pug

```
...

  #navbarSupportedContent.collapse.navbar-collapse
    ...
      if isAuthenticated
        li.nav-item.dropdown
          a.nav-link.dropdown-toggle(href='#', data-toggle='dropdown', role='button', aria-haspopup='true', aria-
expanded='false') Blogs
          .dropdown-menu
            a.dropdown-item(href='/blogs') Blogs
            ...
      if !isAuthenticated
        li.nav-item
          a.nav-link(href='/authors/new') Register
        li.nav-item
          a.nav-link(href='/login') Login
      if isAuthenticated
        li.nav-item
          form.nav-destroy(action=`/logout`, method="POST")
            input.destroy(type="submit", value="Logout")
      ...
```

Using our helper, we can hide the **blog links** unless the user is authenticated. If the user isn't authenticated, we'll show **Register** and **Login** which will provide a nice user experience as they won't show when the user is logged in. Last, we'll show the **Logout** link if the user is authenticated as that would be their only session action at that point.

BREAK

Associations

Adding Associations to the Blog Model

In Mongoose there are 2 ways to associate other documents within a document. We can **nest** them or we can **ref**erence them. Nesting means embedding a schema within a schema. This enforces a structure for a child document but keeps the actual data within the parent document.

Referencing a document is more like the SQL way associations are done. This utilizes the document's identifier (usually _id) to create a reference to the original document. In our **BlogSchema** we want to create a reference to an **Author**as blogs should belong to authors. The syntax may look strange, but it's quite simple:

- **type** is a **mongoose.Schema.Types.ObjectId** - this is because the ObjectId type is what is used to store _ids for a document
- **ref 'Author' -** refers to whichever collection you want to create the association with. In this case, our Author collection
- required - our validation. We must have an author

/models/author.js

```
// We will need our mongoose library
const mongoose = require(`mongoose`);

// Our schema
const BlogSchema = new mongoose.Schema({
  ...,
  author: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Author',
    required: true
  }
...
```

Adjusting Our Queries for Only Logged-in Author

Blogs should be owned by an author. Because of this, we want the author to be able to interact with their blogs only. Any editing/deleting needs to be restricted. Reading could be public, but for now we'll restrict everything to the current logged in author.

You will remember we created an app level helper method that will allow us to check if a user is authenticated or not. If they're not they will be redirected to the home page. We could utilize this action in our **/routes/blogs.js** for a more **DRY** approach, but by utilizing it in the controller we get a lot more control... **req.isAuthenticated()** will stop a user from accessing the blog actions a non-authenticated user is not allowed to access.

.find is a query helper that comes bundled with Mongoose. It is great for getting all the documents in a collection, but it supports a filtering argument as well. Because we have an author id stored in our sessions, we can reference the author id by simply accessing that session property: **req.session.userId**. This will give us a way to filter our result sets to only documents viewable by that author.

If we want to include information from the author document, we have to embed the author document into the returned blog documents. We can use the Mongoose helper method **.populate** to make this much easier. We simply provide it a string argument of the BlogSchema reference attribute we want to embed. We only have one in our schema currently (**author**) but we could easily have more and provide **.populate** more strings. What the end result will be is the ability to access the author record by simply climbing our way through the property chain:
blog.author.firstName

/controllers/blogsController.js

```
const Blog = require('../models/blog');
const mongoose = require('mongoose');

exports.index = (req, res) => {
  req.isAuthenticated();

  Blog.find({
      author: req.session.userId
    })
    .populate('author')
    .then(blogs => {
      ...
};


exports.show = (req, res) => {
  req.isAuthenticated();

  Blog.findOne({
      _id: req.params.id,
      author: req.session.userId
    })
    ...
};


exports.drafts = (req, res) => {
  req.isAuthenticated();

  Blog.find({
      author: req.session.userId
    }).drafts()
    ...
};


exports.published = (req, res) => {
  req.isAuthenticated();

  Blog.find({
      author: req.session.userId
    }).published()
    ...
};


exports.new = (req, res) => {
  req.isAuthenticated();

  ...
};


exports.create = async (req, res) => {
  req.isAuthenticated();

  // Add the current author to the blog
  req.body.blog.author = req.session.userId;

  ...
};


exports.edit = (req, res) => {
  req.isAuthenticated();

  Blog.findOne({
      _id: req.params.id,
      author: req.session.userId
    })
    ...
```

```
};


exports.update = (req, res) => {
  req.isAuthenticated();

  Blog.updateOne({
      _id: req.body.id,
      author: req.session.userId
    }, req.body.blog, {
      runValidators: true
    })
    ...
};


exports.destroy = (req, res) => {
  req.isAuthenticated();

  Blog.deleteOne({
      _id: req.body.id,
      author: req.session.userId
    })
    ...
};
```

You'll notice under **exports.create** we added the line **req.body.blog.author = req.session.userId;**. This is so we can add the **author id** to our blog document. This is essential if we're going to be able to access the author in the future. This also means any current unclaimed blogs (that don't have an author) will no longer be accessible for edit or destruction as they need to be owned by the author for those actions to occur.

Adding the Author's Name to the blogs/index View

The author only has access to their own blogs. it would be nice to add their name to the index view. We could adjust our query in the blogs controller to deliver all the blogs and then only show actions for the one's owned by the author. Either way, seeing the name makes sense, so let's implement it:

/views/blogs/index

```
extends ../layouts/main.pug
block content
  .container
    header
      h1= title

    div
      table.table.table-striped
        thead
          tr
            th Title
            th Status
            th Author
            th

        tbody
          each blog in blogs
            tr
              td
                a(href=`/blogs/${blog.id}`)= blog.title
              td= blog.status
              td= `${blog.author.firstName} ${blog.author.lastName}`
              td
                a(href=`/blogs/${blog.id}/edit`) edit
                = ` | `
                form.destroy(action=`/blogs/destroy`, method="POST")
                  input(type="hidden", name="id", value=`${blog.id}`)
                  input.destroy(type="submit", value="destroy")
```

Interestingly enough, we could create a virtual property (like the passwordConfirmation) called **fullName** that provides the author's name pre-concatenated.