Week 11

INFSCI: 2560 Web Technologies and Standards

Looking forward

- November 5 (W) - Project 3a due (extended)
- November 13 (W)
  - Project 3b due (extended)
  - Activity 8 due (assigned at end of class, today)
- November 15 (F)
  - Final Project teams and project descriptions due
- November 18 (M) - Class (security, oath)
- November 23 (M) - No Class, Thanksgiving Break
- December 2 (M) - Exam 2
- December 9 (M) - LAST class, project presentations

Agenda

- More on Views with EJS
- Database operations & CRUD

Model View Controller (MVC) Review

- **What is the definition of MVC?**
  - Architecture that aims to separate out the data, the display and the application logic.
- **Model**
  - Where the **data** is stored.
  - We will be using MongoDB and the Mongoose ORM/ODM with Express
- **View**
  - How the data is **display**ed to users.
  - We will be using the EJS Template system with Express to display HTML.
- **Controller**
  - The **application logic**. The glue between the model and view.
  - We will be using Routes in Express.

Views and Templates

- Express is an unopinionated framework.
- Supports various view engines
  - EJS - A simple template system that uses JavaScript
  - Pug (formally Jade) - A concise framework for writing HTML
  - Mustache.js - A logic-less template system for generating any kind of text file (HTML, configs, source code) using the Mustache template language.
- View engines (such as EJS) will render a template
- Rendering means to generate content (usually HTML pages) by injecting values into templates

Templating with EJS

- EJS stands for *Embeded JavaScript*
- It is a very simple language that leverages JavaScript to inject values
- EJS has four major features:
  - JavaScript that is evaluated a not printed (for logic and control flow)
  - JavaScript that evaluated, escaped, and printed (for injecting values safely)
  - JavaScript that is evaluated, *not escaped*, and printed (for inserting HTML)
  - Filters for modifying output values in the template
- You write HTML (or other text files) and then put EJS tags into the file to inject data or execute logic

Sending data to a view

- Express builds a *context object* every time you call render
- Context objects are passed to the view engine and injected into variable placeholders in the template
- To send data to your view template you can modify two objects:
  - app.locals - JS object and its properties are local variables within the application. Available to all requests.
  - res.locals - JS object and its properties are local variables scoped to the request. Overrides app.locals.
- Use res.render() - To pass an object to the render function.
  - `res.render('index',{user:"John Smith"})`
- The view engine looks for the template file based on name
- You can leave off file extensions, Express will add them based on default view engine

EJS Tags

- <% - 'Scriptlet' tag, for control-flow, no output
- <%_ - 'Whitespace Slurping' Scriptlet tag, strips all whitespace before it
- <%= - Outputs the value into the template (escaped)
- <%- - Outputs the unescaped value into the template
- <%# - Comment tag, no execution, no output
- <%% - Outputs a literal '<%'
- %%> - Outputs a literal '%>'
- %> - Plain ending tag
- -%> - Trim-mode ('newline slurp') tag, trims following newline
- _%> - 'Whitespace Slurping' ending tag, removes all whitespace after it

EJS Example

```
Hi <%= name %>!
You were born in <%= birthyear %>, so that means you're
  <%= (new Date()).getFullYear() - birthyear %> years old.
<% if (career) { -%>
  <%=: career | capitalize %> is a cool career!
<% } else { -%>
  Haven't started a career yet? That's cool.
<% } -%>
Oh, let's read your bio: <%- bio %> See you later!
```

Given this context object >>

```
{
  name: "Tony Hawk",
  birthyear: 1968,
  career: "skateboarding",
  bio: "Tony Hawk is the coolest skateboarder around."
}
```

EJS Example

```
Hi <%= name %>!
You were born in <%= birthyear %>, so that means you're
  <%= (new Date()).getFullYear() - birthyear %> years old.
<% if (career) { -%>
  <%=: career | capitalize %> is a cool career!
<% } else { -%>
  Haven't started a career yet? That's cool.
<% } -%>
Oh, let's read your bio: <%- bio %> See you later!
```

```
Hi Tony Hawk!
You were born in 1968, so that means you are 51 years old.
Skateboarding is a cool career!
Oh, let's read your bio: Tony Hawk is the coolest
skateboarder around. See you later!
```

Another Example

```
// Make a mapping numbers to names
const days_of_week = [
  "Sunday",
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday"];

// Respond to default route
app.get('/', function(request, response) {

  // Get today's day as number
  let day = (new Date()).getDay();
  // Set response context with current day
  response.locals.day = days_of_week[day-1];

  // Render response with name contxt
  response.render("index", {name: request.query.name});
});

// listen for requests :)
const listener = app.listen(process.env.PORT, function() {
  console.log('Your app is listening on port ' + listener.address().port);
});
```

```
<h1>
  <% if (name) { %>
  Hello, <%= name %> !
  <% } else {%>
  Hello, Stranger !
  <% }%>
</h1>


<p>
  I see that it is a <%= day %>.
  <% if (day === favorite_day) {-%>
    <%= day %> is my favorite day!
  <% } else { -%>
    <%= day %> is a meh day. I prefer <%= favorite_day %>.
  <% } -%>
</p>
```

https://glitch.com/~2560-ejs-demo

Database Definitions

A **database** (**DB**) is an organized collection of data.

- In Project 3a, we used a JSON file to store the dictionary information.
- By this definition, the JSON file can be considered a database.

A **database management system** (**DBMS**) is software that handles the storage, retrieval, and updating of data.

- Examples: MongoDB, MySQL, PostgreSQL, etc.
- Usually when people say "**database**", they mean data that is managed through a DBMS.

Why use a Database instead of saving to a JSON file?

- **fast**: can search/filter a database quickly compared to a file
- **scalable**: can handle very large data sizes
- **reliable**: mechanisms in place for secure transactions, backups, etc.
- **built-in features**: can search, filter data, combine data from multiple sources
- **abstract**: provides layer of abstraction between stored data and app(s)
    - Can change **where** and **how** data is stored without needing to change the code that connects to the database.

Disclaimer

Databases and DBMS is a HUGE topic in CS with multiple courses dedicated to it:

- CS1555: Database Management Systems
- CS2055: Database Management Systems
- CS2550: Principles of Database Systems
- INFSCI 1022: DATABASE MANAGEMENT SYSTEMS
- *and more ...*

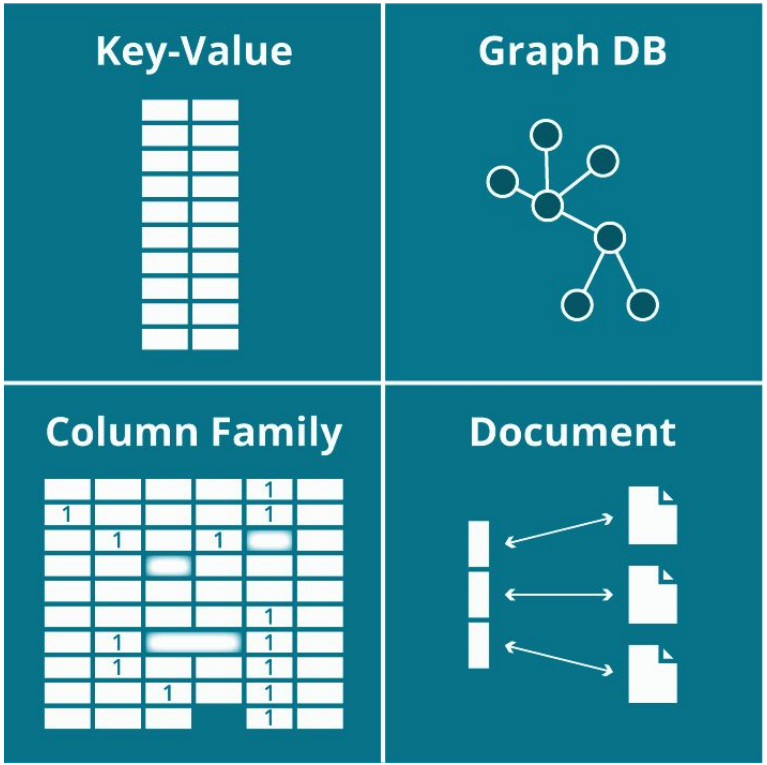
In this course, we will cover only the very basics:

- How one particular DBMS works (MongoDB)
- How to use Mongoose with Express JS to perform CRUD operations.

Types of Databases

- Two main kinds, SQL and NoSQL
- Most Server Side stacks use Relational or SQL databases
- MySQL is a popular open source SQL database
- It is a good idea to learn SQL
- In this class we are going to focus on NoSQL Datastores

Types of NoSQL



Types of NoSQL Databases. Image from Neo4J

- Four kinds of NoSQL Datastores
  - Key-Value - A big hash table of keys and values
  - Graph - A network of nodes as data connected by edges
  - Column - Blocks of data stored as a single column
  - Document - Documents made up of tagged elements
- Picking a NoSQL database type depends on the data and application requirements
- **In this class we are going to use a document database**

Document NoSQL Databases

- A document database is a type of non-relational database that is designed to store semistructured data as documents.
- ***Documents*** are typically represented as a JSON document, but could be XML as well
  - Like a row in an relational database
- Documents are organized into ***collections***
  - Like a table in a relational database
- ***Document databases*** store all information for a given object in a single instance in the database, and every stored object can be different from every other.

Document No SQL Databases

An example document encoded as JSON

```
{
  "firstname": "Bob",
  "address": "5 Oak St.",
  "phone": [ {"work": "(890) 555-0133"},
             {"cell": "(123) 555-0178"}
           ]
}
```

An example document encoded as XML

```
<contact>
  <firstname>Bob</firstname>
  <lastname>Smith</lastname>
  <phone type="Cell">(123) 555-0178</phone>
  <phone type="Work">(890) 555-0133</phone>
  <address>
    <type>Home</type>
    <street1>123 Back St.</street1>
    <city>Boys</city>
    <state>AR</state>
    <zip>32225</zip>
    <country>US</country>
  </address>
</contact>
```

- These two documents share some structural elements with one another, but each also has unique elements.
- **Document databases do not have a *schema***

Relational Databases vs Document Database

Relational databases have fixed schemas; document-oriented databases have flexible schemas

Relational Database

Name	School	Employer	Occupation
Ada	null	Self	Entrepreneur
Tonya	Pitt	null	null

Document Database

```
{
  name: "Ada",
  employer: "Self",
  occupation: "Entrepreneur"
}
{
  name: "Tonya",
  school: "Pitt"
}
```

MongoDB

- MongoDB is the most popular Document-Oriented NoSQL Database
- Stores data as BSON, Binary JSON, and works well with JavaScript

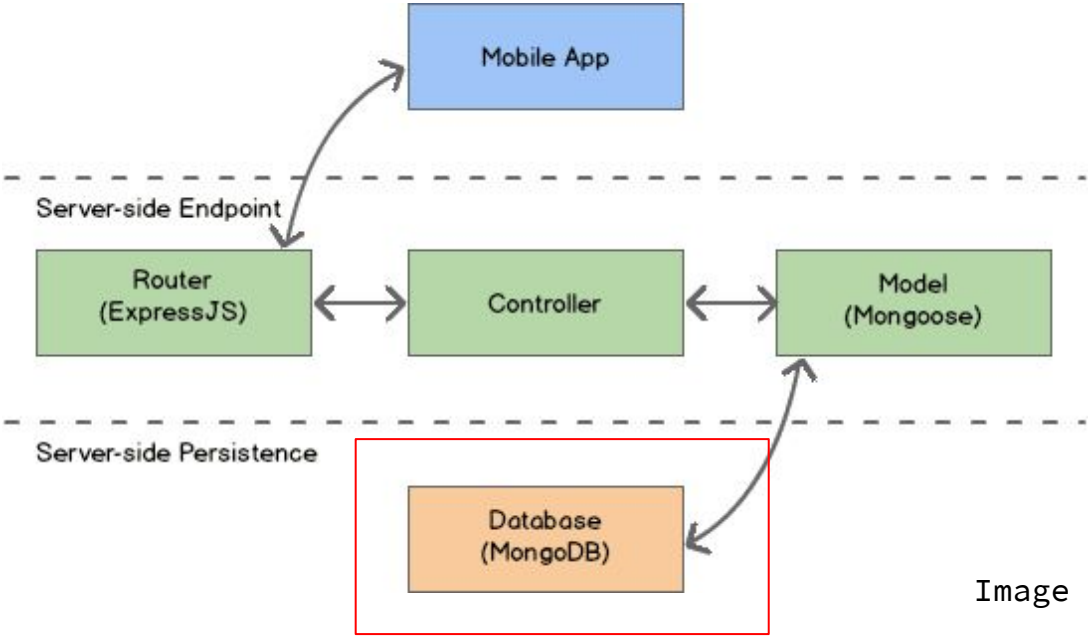


Image from DZone

MongoDB

Collections

‘Collections’ in Mongo are equivalent to tables in relational databases. They can hold multiple JSON documents.

Documents

‘Documents’ are equivalent to records or rows of data in SQL. While a SQL row can reference data in other tables, Mongo documents usually combine that in a document.

Fields

‘Fields’ or attributes are similar to columns in a SQL table.

Schema

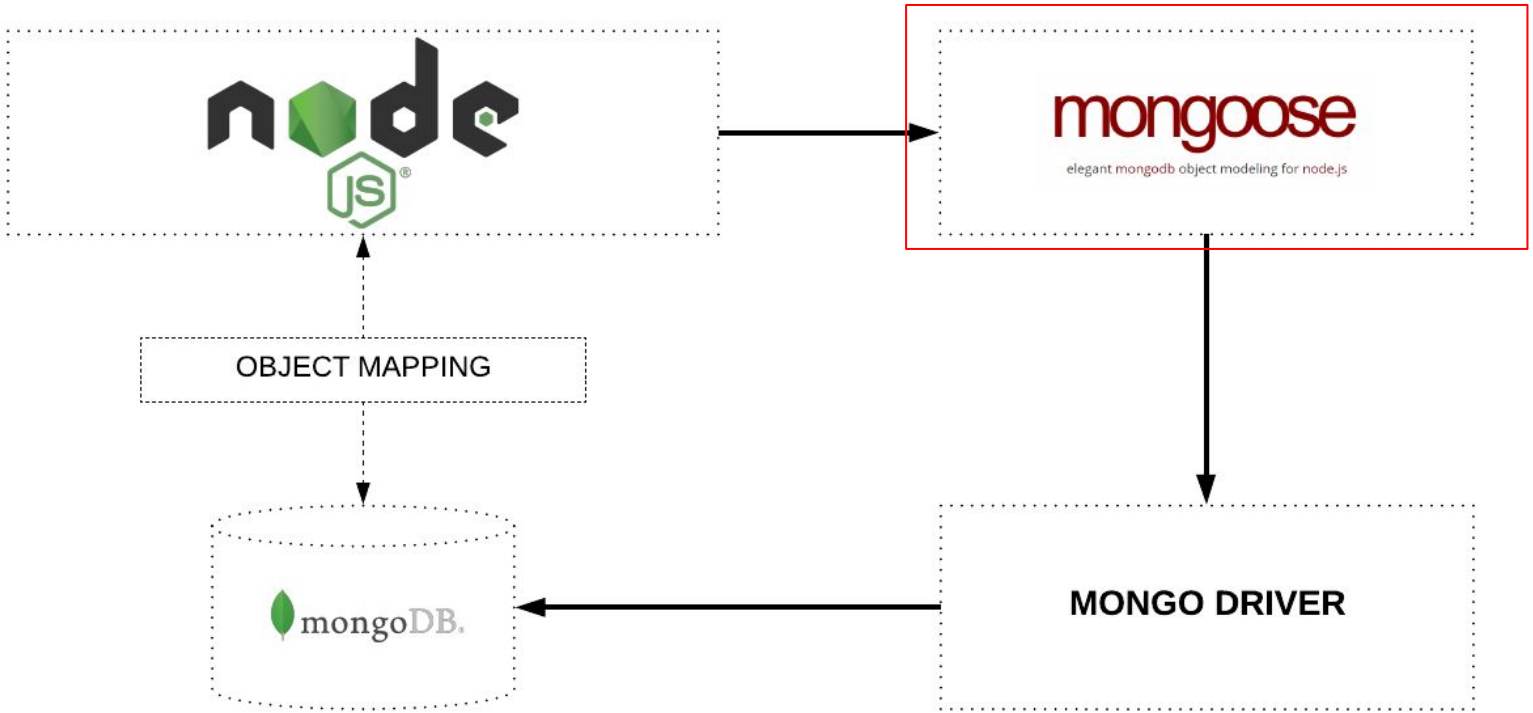
While Mongo is schema-less, SQL defines a schema via the table definition. A Mongoose ‘schema’ is a document data structure (or shape of the document) that is enforced via the application layer.

Models

‘Models’ are higher-order constructors that take a schema and create an instance of a document equivalent to records in a relational database.

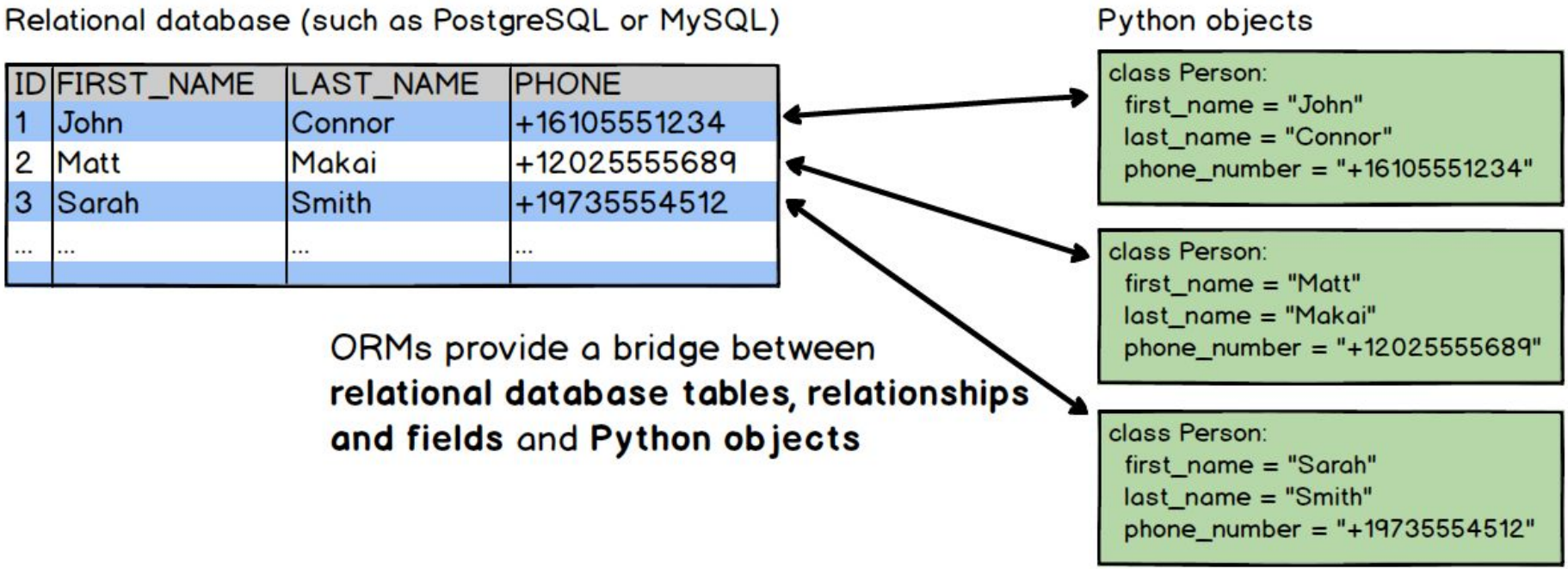
10 minute break

Mongoose - A MongoDB ODM for Node



MongoDB & Mongoose diagram. Image from FreeCodeCamp

Accessing the Database using an ORM (Python Example)



An Python ORM example. Image from Fullstack Python

Create, Read, Update, Delete -> CRUD

CRUD Overview

- CRUD or Create, Read (sometimes retrieve), Update, and Delete are the four basic functions of persistent storage
- Not an official standard, but rather a strong conceptual model
- It provides an abstract way of thinking about database operations across different protocols or query languages
- Useful when designing the architecture of a database driven web application or API

CRUD Overview

Operation	**SQL**	**HTTP**	**RESTful WS**
Create	INSERT	PUT / POST	POST
Read (Retrieve)	SELECT	GET	GET
Update (Modify)	UPDATE	PUT / POST / PATCH	PUT
Delete (Destroy)	DELETE	DELETE	DELETE

Setting up MongoDB & Express

- There are three steps necessary to wiring a MongoDB to an Express application
  1. Establish a connection with MongoDB
  2. Define the data models
  3. Create RESTful route handlers for CRUD operations

Connect To MongoDB

```
// Establish a connection with the Mongo Database
const mongoDB = ("mongodb+srv://"+
                 process.env.USERNAME+
                 ":"
                 +process.env.PASSWORD+
                 "@"+process.env.HOST+
                 "/"
                 +process.env.DATABASE);
mongoose.connect(mongoDB, {useNewUrlParser: true, retryWrites: true});
```

- This code creates a string containing the URL for accessing a MongoDB
- There are sensitive pieces of information so they are stored as private environment variables (with Glitch)
- It is generally not a good idea to hard-code your MongoDB password in your source code.
- See the Mongoose Connection documentation for more information

Define the Data Models

```
// Data Model for Books
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const BookSchema = new Schema(
  {
    title: {type: String},
    author: {type: String},
  }
);

// Export model
module.exports = mongoose.model("book", BookSchema);
```

- By convention, put models in a separate folder and files
- One file per model
- Use the Mongoose Schema to define JavaScript Objects with fields according to the MongoDB datatypes
- See the Mongoose documentation on Schemas and SchemasTypes

Mongoose Schema Validation

- Mongoose has several built-in validators
  - Numbers have min and max validators
  - Strings have enum, match, minlength and maxlength validators.
- The unique Options is not a validator
  - This is a helper for building MongoDB unique indexes

```
const BookSchema = new Schema(
        {
          bookId: {type: String, unique: true},
          title: {type: String},
          author: {type: String},
        }
```

```
var breakfastSchema = new Schema({
  eggs: {
    type: Number,
    min: [6, 'Too few eggs'],
    max: 12
  },
  bacon: {
    type: Number,
    required: [true, 'Why no bacon?']
  },
  drink: {
    type: String,
    enum: ['Coffee', 'Tea'],
    required: function() {
      return this.bacon > 3;
    }
  }
});
```

https://mongoosejs.com/docs/validation.html

Route Handlers

```
// Route handlers
const express = require('express');
const router = express.Router()

//import data models
const Book = require("../models/book");
```

- Use Express.js route code, but use Mongoose inside the functions
- You first load the models and use those objects for accessing the datastore
- The models you created handle all of the communication with the database
- See the Mongoose documentation on Documents for more information

CREATE documents in the Database

```
//CREATE
router.post('/', function(req, res){
  let book = new Book(req.body);
  book.save();
  res.status(201).send(book);
});
```

The result is a document that is returned upon a successful save:

```
{
  _id: 5a78fe3e2f44ba8f85a2409a,
  author: 'Tonya Edmonds',
  Title: 'My Famous Book on Web
Standards',
  __v: 0
}
```

- You use POST requests to endpoints to create new objects/documents
- Inside the POST request handler you create a new object from the data in the POST request
- Use the .save() method to save that object to the database
- This will return the JSON with the newly created object

See the Mongoose Model documentation for more information

READ documents in the Database

```
// RETREIVE all books
router.get("/", function(req,res){
  Book.find({}, function (err, book_list){
    res.json(book_list);
  });
});

// RETRIEVE a specific book
router.get("/:bookId", function(req, res){
  Book.findById(req.params.bookId, function(err, book) {
    res.json(book)
  });
});
```

- These two route handlers retrieve a list of all the documents in the database and grab a specific book based upon an ID in the URL path
- The .find() method will return a list of documents as JavaScript objects
- The .findById() method will return a single document based upon the MongoDB id specified
- See the Mongoose documentation on Queries for more information about reading data from the database

UPDATE documents in the Database

```
//UPDATE
router.put("/:bookId", function(req, res) {
  Book.findById(req.params.bookId, function(err, book) {
    book.title = req.body.title;
    book.author = req.body.author;
    book.save();
    res.json(book);
  });
});
```

- PUT requests with a document ID are used to update document fields
- The body should contain the complete new document
- Use the PATCH HTTP method to do partial updates (not shown)

DELETE documents in the Database

```
//DELETE
router.delete("/:bookId", function(req, res){
  Book.findById(req.params.bookId, function(err, book) {
    book.remove(function(err){
        if(err){
          res.status(500).send(err);
        }
        else{
          res.status(204).send('removed');
        }
    });
  });
});
```

- DELETE requests with a document ID are used to remove documents from the database
- This uses the .remove() deletes a document from the database based on id. There is also the .deleteOne() method to delete based on queries
- This doesn't have any graceful error handling so the bad queries don't respond very well

Glitch Project

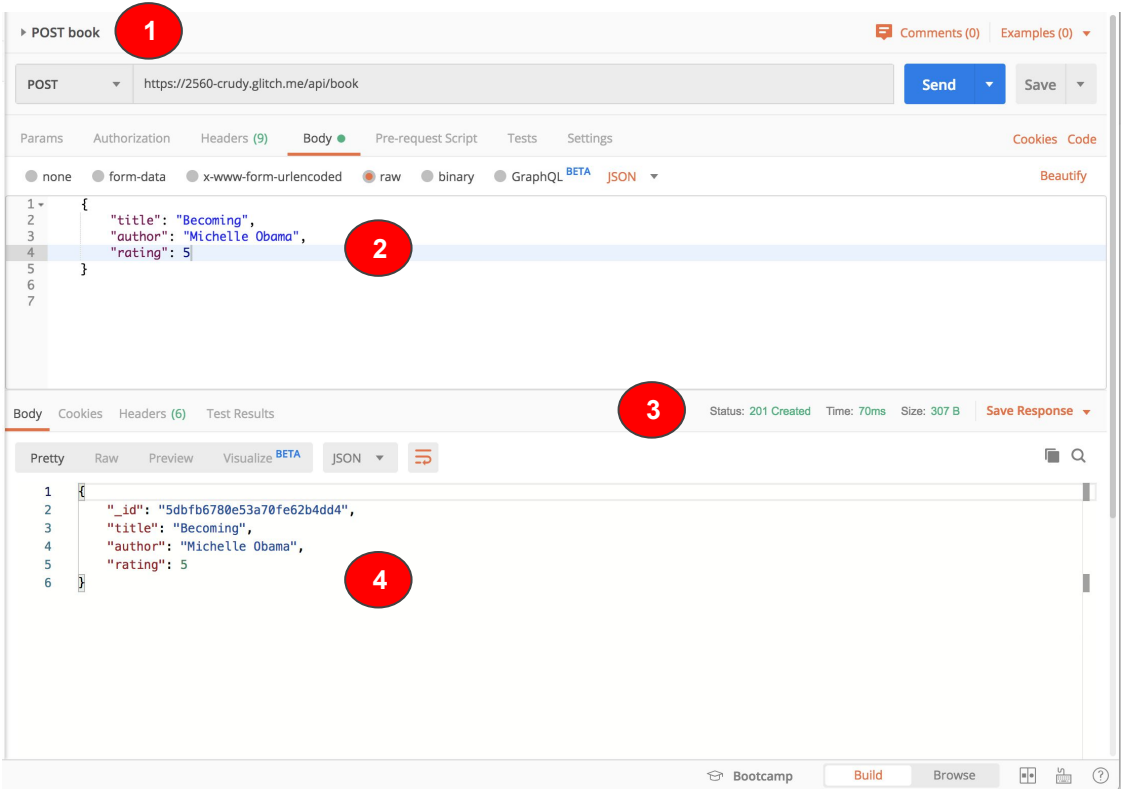
MongoDB and Mongoose References

- See Week 10 Resources

Project 3b

Due November 13 by 11:59p (extended)

POSTing with Postman

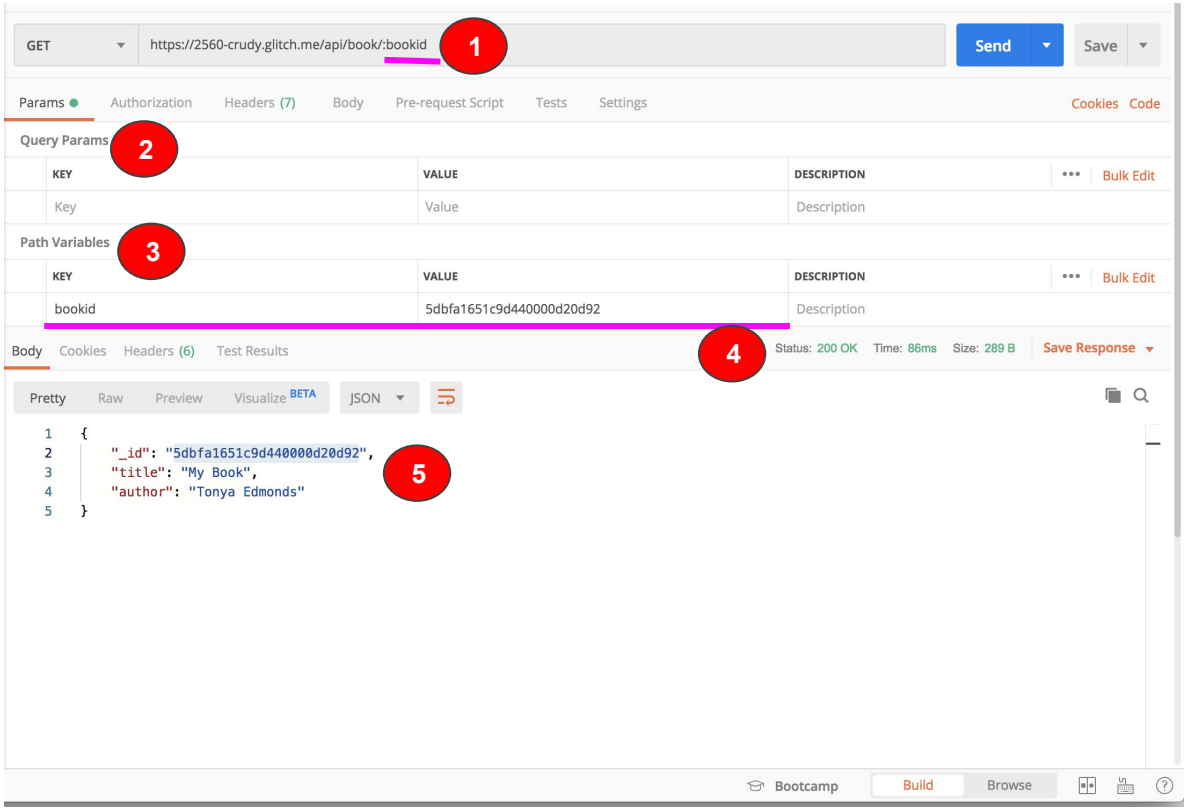


1. Make sure the operation is set to POST. Enter the URL/endpoint.
2. Add the message body. Be sure to select the correct data type

Click Send

3. The POST Status
4. The POST response

GETing data with POSTMAN



1. URI with variable for path parameter
2. Query parameters
3. Path Parameters
4. Response Status/Code
5. Response Body

[Postman Video Tutorials]
[Postman Tutorial]

Activity 8

In this activity you will setup a MongoDB account, which you will need to complete Project 3b.

Due: Wednesday, November 6 by 11:59p

Link