acm.hack

StackSchool

Contents

Introduction to Full Stack 1

"There are two mistakes one can make along the road to truth: not going all the way, and not starting."

– Alan Cohen

As a beginner, full stack development can be quite intimidating. Looking it up yields a seemingly endless list of languages and technologies that you must learn in order to even get started, and who has time for that? The good news is that we can drastically limit the number of things to learn by making a couple of strategic decisions before we start. The bad news is that you will still have to learn about six technologies, three of which we're going to assume you have at least a basic understanding of going into the workshop series. That being said, stick with us! By the end of this, you'll have no trouble making your own full stack application to rival Facebook or Twitter and you'll have a lot of fun doing it.

Before we get into any actual content, allow me to first give you some advice on approaching this series[1]. At first, it's not going to be easy. It'll be frustrating and you'll bang your head against the wall and you'll want to quit. You'll want to quit often. But don't. Stick with it, and eventually you'll make progress. You'll figure out what was causing that unreadable error or that 400 response code, and the feeling of accomplishment will be like no other. Little victories will begin to pile up around you and, before you know it, you'll have a completed full stack app ready to show off to the world. It won't be easy, but nothing worth doing ever is. And keep in mind you're not on your own here! You have a team of twenty passionate, wonderful Hack officers ready to help you through your struggles. All you need to do is reach out![2]

1: And I guess learning in general.

2: Which you can do in person or on discord!

Another piece of advice on learning: it's often helpful to start with an understanding of surface level concepts and dig deeper once you have those mastered.[3] This is the approach we will mostly take in this workshop series. Our weekly workshops will provide a high level view of things in order to get you started, offering occasional nuggets of deeper insight, while the textbook will usually be the place to go if you want a deeper understanding. I highly recommmend utilizing both if you truly want to learn the material. If you would prefer, you can also skim the textbook ahead of our workshops and get an idea of the content that way. Whatever works best for you! Throughout the series, we'll show the entire process of building a simple full stack app. By the end, hopefully you'll be able to make your own!

3: Keep in mind there are many different kinds of learners, so what works for one person may not work for another and vice versa. This is just a suggestion!

1.1 Prerequisites

As mentioned, due to time constraints we unfortunately won't be able to cover absolutely *everything* you need in order to make a full stack app. In

particular, we'll be assuming a basic understanding of frontend concepts. This includes:

- Fundamental Coding Concepts (Think CS31)
- Basic Shell Commands (`cd`, `ls`, etc.)
- HTML/CSS
- Javascript
- React

Luckily, we had an entire workshop series[4] covering these ideas last quarter if you need to catch up! We cover these concepts every year in Hackschool, so check out those workshop recordings/slides/README's before attending. We will be giving slight refreshers on these concepts here, but they'll be very quick and likely difficult to follow if you've never seen them before. Alright, with that out of the way... let's get to the content!

4: Check out our workshops archive here.

1.2 Why Full Stack?

As any good student should, you may be wondering: *What's the point?* To illustrate that, let's take a look at a website that you're probably familiar with.

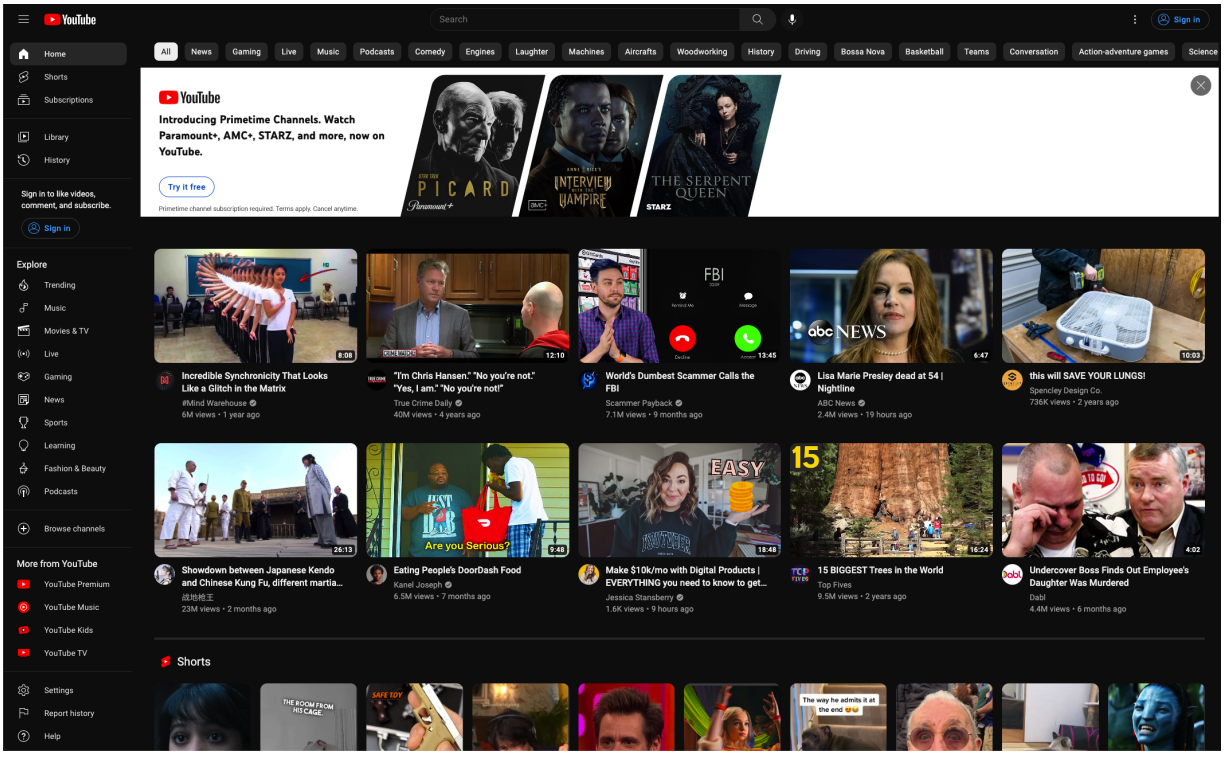


Figure 1.1: A screenshot from YouTube's home page.

If we had to design YouTube from scratch, how might we do it? The most obvious component is the UI. It's likely that some mix of HTML/CSS is used in order to deliver the experience you see in the screenshot. There are several other things to think about, however. For instance, where do the videos come from? They're clearly not all hardcoded into the website

as that would be a nightmare to maintain. You'll also notice that when you refresh the page, YouTube will adjust which videos it recommends to you to ensure it never gets stale. It takes things a step further if you sign in, tailoring its recommendations based on your previous viewing history. How are they doing this?

Unfortunately, making a website as intricate as YouTube is impossible without incorporating the full stack. In order to store all the videos and users, something called a **database** is used. Recommendations are determined by an algorithm on a **server** and communicated to the **client** via an **API**. We'll go over exactly what all of these terms mean, but for now just know that they are each essential components of a full stack application. Each of these components works together in order to deliver a product that millions of people use everyday! And it's not just YouTube. The vast majority of websites and apps you frequent[5] utilize all of these components in order to give you the best experience possible.

5: Google, Instagram, Twitter, Facebook, BeReal. All of these are full stack applications!

Note that not all websites require the full stack. These websites are known as **static sites**, due to the fact that their content is the same for every user that visits and will not change unless the website itself is updated. One example of such a site is Hack's website!

1.3 What is Full Stack?

Fullstack refers to all of the technologies that are needed to complete a project. Typically, we consider full stack development to be composed of two main components called the **frontend** and **backend**. Naturally, this begs the question: what's the difference?

Frontend

Frontend is essentially everything that the user interacts with. Think of everything that you see when you interact with a website: a text box, the colors, buttons, links, navigation bars, pages, etc. Typically, the front end is coded in HTML, CSS, and JavaScript. For our demo, we will be using React.js, which is a JavaScript library that helps us with creating our frontend. Essentially, React provides us with lots of useful library functions and integrations that abstract away many of the tedious parts of web development.[6]

6: Such as DOM interactions. Gross.

Backend

If we look at full stack development as an iceberg[7], the frontend can be considered its tip. What remains deep underwater is the backend. In contrast to frontend development, backend is essentially everything that the user **does not** see. For questions like "*where does the data that I put into my text box go?*" or "*how does my website authenticate me as a proper user?*", we turn to the backend. The backend is made up of multiple parts as well. The ones that we will be focusing on are the server and the database. In order to introduce you to these technologies, let's take a look at them from a very high level.

7: Which for some reason is a popular comparison.

What's a server?

A server is essentially a computer or program that serves as the *brains* of your application. It handles everything that our app needs in order to function, such as interacting with our database, computationally intensive tasks, user authentication, and more. There are many technologies available for servers, but here we will discuss a technology called Node in conjunction with a library called Express. It's here that we define our backend API, which is the interface between our frontend and backend. Servers can also be referred to as backend applications.

What's a database?

A database is just an organized collection of data. It abstracts away the storage and retrieval of data in order to make our lives as full stack developers easier. There are several design philosophies surrounding databases, but two of the most popular are relational and non-relational (also known as NoSQL) databases. We'll briefly cover relational databases here with an emphasis placed on the non-relational side.

This was all super high level, but don't worry! We'll be going over each of these in much more detail in chapters to come.

1.4 MERN

For our workshop series, we will be using the M.E.R.N. stack. Each of these letters stand for a specific technology.

- MongoDB: A NoSQL database that stores all of our persistent data that the app needs to function, such as users and posts.
- Express: A backend web application framework.
- React.js: A frontend JavaScript library that will help us create our frontend through several useful abstractions.
- Node.js A Javscript runtime environemnt used for creating servers.

Downloads

Before getting started, make sure you set up the required technologies!

- Download and install a text editor: VS Code
- Download and install Node.js: Node.js
- Download and install Yarn (our package manager of choice): Run the following command after installing Node: `npm install -global yarn`[8]
- Create a MongoDB account: MongoDB

8: Or use corepack.

You're now ready to get started!

Slides
The slides for this chapter can be found here.

Databases 2

"It is a capital mistake to theorize before one has data."

– Sherlock Holmes

As of the time of this writing, humanity has created nearly 100 zettabytes[1] of digital data. Strongly tied to recent developments in Internet of things and machine learning, familiarity with data storage is a useful tool for any developer. But before delving into the methods and challenges associated with storing data, it's important to fully address why external data storage is so useful.

To do so, we can consider an analogy of a restaurant serving breakfast. Let's assume a new customer enters the restaurant, orders some eggs and bacon, and sits down to eat. The waiter then communicates this order to the kitchen, where a low ranking cook will rush to gather the eggs and bacon required from the fridge in order to prepare the dish. A second customer then enters and attempts to order some eggs, only to be told by the waiter that the restaurant is short on supplies. Why might this be the case? Sharp readers will know that it's likely that the last of the eggs in the fridge were used to make the first customer's dish! In this scenario, the fridge represents our **database**, the eggs represent our **data**, and the restaurant dining room along with the customers inside represent our web application. From this example, we note that creating a shared supply of data for many users necessitates an external database to keep things consistent between them. Just as it's not possible for a restaurant to simultaneously *have* and *not have* eggs in their fridge, we need to keep the data for our web app in a centralized location to allow for a degree of consistency.

1: That is, 8×10^{21} bits. To put that into perspective, the typical hard drive is 1 terabyte in size. It would take 100 billion of these to store all of this data.

In addition to facilitating a shared experience between users, aggregating data in a database solves another issue: data persistence. **Data persistence** refers to the concept of any changes made by a user in an application continuing to exist, even if the application is closed or restarted. We can liken this to a customer entering a restaurant and buying their last ham sandwich on Monday, then returning on Tuesday to find the restaurant out of ham. The change made to the state of the restaurant persists into the next day, ensuring continuity in the customer's experience.[2] Databases can be utilized to accomplish this in the context of a web app.

2: Even though it may not be what the customer was hoping to see upon arrival.

With a strong motivation for utilizing a database, let's delve deeper into what databases actually store, common implementations, and how to interact with them.

2.1 Spreadsheets and Schemas

Databases can be as simple as a set of flat files in our file system. Assuming our web application has read and write access to our files[3], we

3: Giving a program read access to a file allows it to view its contents while write access allows it to modify the contents of the file.

can persist any changes even if our web application has closed. However, this implementation can be challenging to work with and is a bit more granular that we will need for our tech stack.[4] Instead we will begin by considering its cousin, the **spreadsheet**. A spreadsheet is a file with columns and rows - effectively a large table.[5] In our examples we treat each row as an individual entry in our table, and recognize each column as a a unique property of our data. Common examples of spreadsheet file types are the comma separated file format (.csv), and Microsoft Excel's various formats (.xls, .xlsx, .xlsm).

4: Fun fact: up until about 15 years ago, UCLA kept all student records in flat files. However, this system had many issues due to poor data integrity and inconsistencies, leading to a switch to a more traditional database.

5: You can actually generate these without using any external libraries. Try to think how you might encode a table and its data in a text file.

Item	Quantity	Price
Lettuce	20	3
Tomatoes	20	3
Broccoli	30	4
Carrots	80	3

Let's refer back to our previous restaurant scenario. Now, instead of walking into the fridge to find and keep track of any raw ingredient, the restaurant chef can look at a note posted on the fridge that lists each item contained within. The table above displays a simple example of such a note. The set of column headers[6] for this table is known as the table **schema**, representing the data type and name associated with the values along each row in the table.

6: In this case: Item, Quantity, Price.

As the kitchen staff remove items from the fridge to fulfill orders, we expect the quantity fields of our fridge table to decrease. Likewise, when the restaurant receives a fresh shipment of groceries we expect the associated quantity fields to increase. On the off chance that new ingredients are required, additional rows can be added to the table to display the appropriate properties for that ingredient.

Getting Meta...

In addition to the data we've elected to store, most modern databases have options to include additional columns in a table's schema known as metadata. **Metadata** refers to "data about data," and most commonly appears as a creation or modification timestamp and username, or a version id. The table below shows additional columns `Last_Updated` and `Update_By`, both of which are considered metadata.

Item	Quantity	Price	Last_Updated	Update_By
Lettuce	20	3	12/25/2022	James
Tomatoes	20	3	1/3/2023	Nathan
Broccoli	30	4	1/5/2023	Nareh
Carrots	80	3	12/20/2022	Kevin

2.2 Relational Databases and SQL

The most widely used form of a database is known as a **relational database** - one or more tables each with a fixed schema. This builds upon our previous model of a spreadsheet file and instead now makes use of an application to optimize data retrieval and modification, while still

having the underlying representation. A defining feature of all relational databases is their guarantee to be **ACID** compliant, with ACID being an acronym:

- Atomic: Any operation either fully succeeds or fails and leaves the database unchanged
- Consistent: Any operation to modify data will leave data in a valid state
- Isolated: Any operation is carried out fully independently of any other operation
- Durable: Any operation, once successful, will be persisted in the database.

ACID compliance eliminates common issues stemming from concurrent reads and writes to a single table, and ensures the database is always in a valid state[7]. Another key feature shared by most popular relational databases is their use of **Structured Query Language (SQL)** to interface with the database.

7: When working with an ACID compliant database, we can often run many operations as a single transaction, ensuring our data can be reverted if our operation fails in the middle.

Working with SQL

SQL is the language of choice for nearly all relational databases, offering a way to view, edit, and delete data. Note that we won't be using it in our app's implementation, but its good to be familiar with it for the future. Assuming our table in our database is named "Fridge", below is an example query that generates the following table.

```
SELECT Item, Quantity, Price FROM Fridge
```

Item	Quantity	Price
Lettuce	20	3
Tomatoes	20	3
Broccoli	30	4
Carrots	80	3

To select all columns in our table we simply need to specify the remaining columns, or make use of the wildcard operator, (*), which is shorthand for all columns in the table.

```
SELECT * FROM Fridge
```

Item	Quantity	Price	Last_Updated	Update_By
Lettuce	20	3	12/25/2022	James
Tomatoes	20	3	1/3/2023	Nathan
Broccoli	30	4	1/5/2023	Nareh
Carrots	80	3	12/20/2022	Kevin

Now considering the case of a chef removing items from the fridge to prepare a meal, we make use of SQL's `UPDATE` and `WHERE` keywords, which lends itself to the following syntax. Note the single equal sign = acts as an assignment operator after the `SET` keyword, but ordinarily performs an equality comparison. We also see that SQL's string type is called `VARCHAR`, referring to a variable length character field, and must be quoted with single quotes. The query below will subtract 1 from the carrot and lettuce quantity fields.

```
UPDATE Fridge
SET Quantity = Quantity - 1
WHERE Item = 'Carrots' OR Item = 'Lettuce'
```

Finally, to add a new ingredient to our table, we can make use of the `INSERT` SQL operation. In the case of adding a single row we choose to use the `VALUES` variant, but can opt to use the `SELECT` variant if adding multiple rows. To set the `Last_Updated` metadata field, we make use a database specific date function (`CURDATE()` is built into MySQL). When designing our table, it would be best practice to provide this function as the default value for the Last_Updated column.

```
INSERT INTO Fridge (Item, Quantity, Price,
Last_Updated, Updated_By)
VALUES (
    'Spinach',
    100,
    2.5,
    CURDATE(),
    'ACM Hack'
)
```

After all of our queries, we are left with the following table.

Item	Quantity	Price	Last_Updated	Update_By
Lettuce	19	3	12/25/2022	James
Tomatoes	20	3	1/3/2023	Nathan
Broccoli	30	4	1/5/2023	Nareh
Carrots	79	3	12/20/2022	Kevin
Spinach	100	2.5	1/9/2023	ACM Hack

Fortunately SQL has a very gradual learning curve for its fundamental features, making it relatively simple to jump into using it!

Beyond these features, relational databases support more advance table configurations such as constraints,[8] primary keys, and foreign keys. It is from these features - specifically the ability to enforce relationships between different schema fields and tables - that this type of database derives its name. Below are the five most popular relational databases

1. Oracle
2. MySQL
3. Microsoft SQL Server (MSSQL)
4. PostgreSQL
5. IBM DB2

8: Adding constraints to our database allows us to catch human errors before they snowball into larger issues

2.3 Non-Relational Databases

An alternative (and a better choice in certain scenarios) is a **non-relational database**, frequently referred to as a **NoSQL database**[9]. Like the name

9: Some non-relational databases can be queried with SQL and SQL like languages, but most are intended to be interacted with directly through code.

implies, non-relational database are structured differently from their relational counterparts.

Non-Relational Storage

A key distinguishing feature of a non-relational database is its lack of a rigid schema and of tables altogether. Instead we store each record of data either as a **document** or a **key-value pair**. To understand the meaning and motivation behind this storage method, let's return to our restaurant scenario. Instead of displaying a table on our Fridge with information about our raw ingredients, let's instead display each ingredient on its own distinctly colored sticky note, like in Figure 2.1.

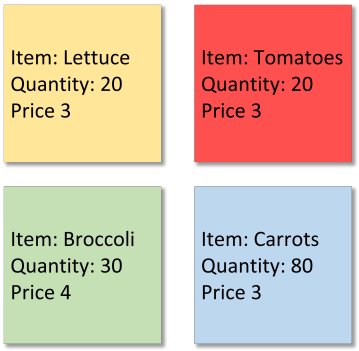


Figure 2.1: A visual representation of a non-relational database

This representation aligns closely with a non-relational database, with each sticky note being analogous to a single document in our database. Note that although in our example, each sticky note contained the same fields, we could add additional fields as desired. We can call a set of documents in a non-relational database a **collection**, a concept akin to a table in a relational database.

Now imagine when the chef of our restaurant has been using this sticky note system for the last couple months and has gotten very used to it - so much so that they simply consider the sticky note color and quantity value before removing the relevant ingredients they need. Here, the sticky note color parallels the concept of a unique id field associated with each document. Similar to the chef's instinctive association of a particular color with a particular document, a non-relational database makes use of a **hash function** to map a document id to a particular document. Although the exact mechanics of these hash functions[10] are beyond our scope, they efficiently associate an id with a document, providing the backbone for fast operations in a non-relational database. For those familiar with the concept (or those planning to take CS 32 in the future), this makes our collection a variant of a **hash table**.

10: Hash functions are generally are implemented with hardware or optimized and concise software and map a unique set of input into a unique id.

Recording Documents: JSON

In most non-relational databases (including MongoDB[11]) documents are written using the JavaScript Object Notation or **JSON**. JSON syntax attempts to provide an efficient way to list a series of key value pairs. Below is a small piece of JavaScript code that would represent the lettuce document in our restaurant example.

11: The 'M' in MERN stands for "MongoDB" - a popular non-relational database

```
lettuce = {
    _id: 00000123456879,
    __v: 0,
    item: "Lettuce",
    quantity: 20,
    price: 3,
}
```

As the name implies, JSON is just a way of encoding a Javascript object as a string. We can encode the object above as follows[12]:

12: Note that a single JSON string corresponds to a single object (though we can group multiple together by nesting objects)

```
    {
        "_id": 00000123456879,
        "__v": 0,
        "item": "Lettuce",
        "quantity": 20,
        "price": 3
    }
```

In our example above, the syntax for each key-value pair is to separate the two with a colon and to comma separate each entry in any given document. The metadata fields included begin with an underscore to signify they are special fields (although the specific format will depend on the database being used). As we will see when constructing our app, modifications and additions to our data will also be formatted in JSON.

The five most popular non-relational databases currently used are as follows

1. MongoDB
2. Redis
3. Elasticsearch
4. DynamoDB
5. Neo4j

2.4 Selecting and Hosting a Database

Although we've elected to use MongoDB, a popular non-relational database as a part of the established MERN tech stack, we will briefly discuss the advantages of each type of database and how to host them. Below are the main advantages of using a relational database

- Integrity: constraints on data can be easily set and enforced
- Reliability: simple and efficient to backup data
- Security: decades of strong and reliable storage methods
- ACID Compliant: any operation on the database will be Atomic, Consistent, Isolated, and Durable
- Transactable: can perform a series of complex operations as a single transaction (good for Online Transaction Processing - OLTP[13])
- Programmability: can define stored procedures and functions to streamline queries

13: Whereas an OLTP use case aligns well with a relational database, a non-relational database will likely also perform well. For Online Analytical Processing (OLAP) - performing analysis on very large datasets - a better choice would be a relational data warehouse.

Non-relational databases provide the following advantages

- Scalability: can scale storage size and compute power
- Open Source: source code of common databases can be analyzed and understood
- Efficient: can perform queries faster than relational databases
- Data Complexity: can store complex data types
- Simplicity: simple configuration and maintenance

As we expect to have a small number of users, would prefer a faster configuration, are not really concerned with ACID compliance or security, and would like efficient code, we choose to use a non-relational database.

It is also important to consider the computer we will run our database on. If a single developer is designing a database, it is possible to run most types of databases locally on Windows, Linux, or MacOS, saving the underlying data on your computer's local disk. Under this setup, the database will be inaccessible when the computer has been shut down, making development by other collaborators challenging.

An alternative to hosting a database on your computer may be running the database on a dedicated computer you can guarantee will be continuously running. After configuring the connection between your web application and database computer,[14] we alleviate the issue of having our data continuously available, but introduce a new issue: we must now maintain a distinct computer dedicated for our database.

14: A computer used to run a database in this way is commonly called a database server. We'll talk more about servers in chapter 3.

A popular alternative that facilitates collaboration is to run the database on a third-party computer with the power of **cloud computing**. Most major cloud service providers offer free storage options, enabling users to run and access a database on a computer in one of their warehouses. To make this process simpler, MongoDB has partnered with AWS, Azure and Google Cloud to offer their **MongoDB Atlas**. By creating a MongoDB Atlas account, one can easily create a database on the cloud, and connect to it from in the browser, through the command line, or through code.

2.5 Mongoose and Asynchronous Programming

The final step of using our hosted database is writing some code to access and manipulate our database. Here we will take advantage of a programming library to make our job a bit simpler.

Using Mongoose

In our restaurant analogy, let's assume there is one kitchen worker named Cameron whose sole job is to retrieve ingredients from the fridge (which can be quite complicated due to its size and eccentric method of organization). Without Cameron, each chef would be required to learn how to navigate this unwieldy fridge in order to get the ingredients required for a recipe. Thankfully, Cam is here to shoulder the burden. In our MERN tech stack, Cameron is akin to the **mongoose** library, a JavaScript Object Data Modeling library for MongoDB that simplifies the process of retrieving and modifying data in our database.

Using mongoose allows us to define an expected schema for the documents in our collection. Let's consider what creating a schema for our documents in our restaurant scenario would look like. First, we must signify our intention to use mongoose, done in the first line of the below code. We then define a new Schema, with each intended data field mapping to its relevant data type and properties. Finally, we choose to export our newly defined model so we can use it in future files.

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

const ingredient = new mongoose.Schema({
        item: {
            type: String,
            required: true
        },
        quantity: {
            type: Number,
            required: true
        },
        price: {
            type: Number,
            required: true
        }
});

module.exports = mongoose.model('Ingredient', ingredient);
```

The schema above corresponds to a Javascript object of the following form:

```
ingredient = {
    item: "beef",
    quantity: 21,
    price: 5
}
```

Mongoose also offers a wide variety of intuitive functions to manipulate and add new documents to the collection with a particular schema.[15] Below is a short list of some of the most commonly used functions, the most relevant of which we will use in our demo.

- `Model.findById()`: Finds a single document by its _id field.
- `Model.findByIdAndDelete()`: Finds a single document by its _id field and deletes it.
- `Model.findByIdAndUpdate()`: Finds a single document by its _id field and updates it.
- `Model.find()`: Finds documents matching certain fields. A more general form of `findById()`.
- `Model.create()`: Creates a new document.
- `Model.count()`: Counts number of documents that match an object in a collection.

15: These functions are part of mongoose's Application Programming Interface (API). An API is simply a term used to describe an interface to take advantage of the features of a programming library, application, or framework.

Asynchronous vs Synchronous

In order to discuss Mongoose in more depth, we must first make a slight detour and discuss a concept that may be entirely new to you: asynchronous programming.[16] Yet again, let's go back to the the restaurant scenario, this time considering a chef, Manny, preparing a dish: ramen with veggies! Veggie ramen is relatively simple to make, but there are a couple of steps one needs to follow:

1. Boiling Water
2. Adding Ramen
3. Cutting Veggies

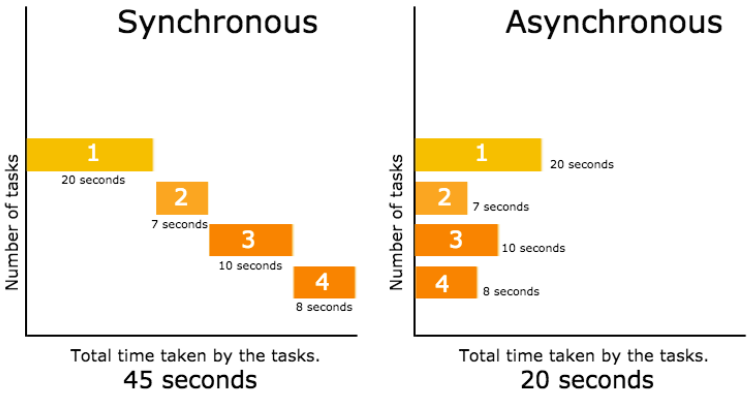
16: Or maybe not. But it's still important to understand asynchronous programming in depth, so stick with us!

4. Adding Veggies

Let's say boiling water takes three minutes and cutting your vegetables takes two. Each of the other steps only takes a handful of seconds, which is neglible for this context. Starting with step 1 and ending with step 4, it would take Manny a total of five minutes to finish preparing the ramen. This is *decent*, but Manny works at a Michelin Star restaurant and he has a reputation to uphold! Rather than doing everything sequentially, he notices that some of the steps are independent of each other and can be done at the same time.[17] With this insight in mind, Manny sets the water to boil and soon after starts cutting the veggies. He finishes cutting the veggies after 2 minutes, and sets them aside. A minute later the water finishes boiling, and so he adds the ramen and the veggies to the pot. Voila! Manny shaved a whole two minutes off of his preparation by deciding to perform some of the steps *at the same time*.

17: Namely, boiling water and cutting veggies.

These two methods of preparation respectively represent synchronous and asynchronous methods of programming! Take a look at the visualization below for a clearer idea of the concept.



Unless otherwise indicated, most of the code you write is synchronous. That is, if there are any tasks that take a long time to complete,[18] they may serve as a bottleneck to other, potentially unrelated tasks. As you might expect, we can get around this with asynchronous programming.

18: Such as getting a lot of data from a database *cough cough*.

Taking things slowly...

Continuing with our analogy, let's write some code that represents the situation. Suppose we have a function for each task mentioned above.

```
1   const boilWater = () => {
2       // implemenation details here
3       // ALWAYS TAKES 3 MINUTES
4       // logs to console when finished
5   };
6
7   const addRamen = () => {
8       // implemenation details here
9       // INSTANT, but can only be done after water finishes
    boiling
10      // logs to console when finished
```

```
};

const cutVeggies = () => {
    // implemenation details here
    // ALWAYS TAKES 2 MINUTES
    // logs to console when finished
};

const addVeggies = () => {
    // implemenation details here
    // INSTANT, but can only be done after water finishes boiling
    // logs to console when finished
};
```

The synchronous approach is as follows and, as noted before, will take five minutes to complete.

```
boilWater(); // 3 minutes
addRamen();
cutVeggies(); // 2 minutes
addVeggies();
```

How can we do better? Introducing promises! **Promises** are objects in Javascript that "represent the eventual completion (or failure) of an asynchronous operation, and its resulting value." In (slightly) more concrete terms, they are wrappers for values that might not yet be determined, which is typical of asynchronous code. Consider the `boilWater()` function. Below is an idea of what an asynchronous version of the function might look like in Javascript.[19]

19: The syntax may seem a bit strange upon first glance and admittedly, it is. The good news is that in the context of this series, this is the only time we will write asynchronous code this way. So don't worry if this is a bit confusing to you!

```
const boilWater = () => {
    return new Promise((resolve, reject) => {
        // implemenation details here
        // ALWAYS TAKES 3 MINUTES
        // logs to console when finished
    });
};
```

We can do the same for our other intensive task, `cutVeggies()`. Now, when we call either of these functions, it won't halt the rest of our code!

```
boilWater(); // 3 minutes
addRamen();
cutVeggies(); // 2 minutes
addVeggies();
// in total, will only take 3 minutes... but there's a problem.
```

Unfortunately, we have an issue. As the code is currently written, we do not distuinguish dependent and independent tasks. This means that we will end up adding pasta to our water before we finish boiling and we'll add our veggies before we finish cutting. How can we handle this? We'll show two equally viable ways.

1. Using `.then()`

```
cutVeggies(); // 2 minutes
boilWater().then(() => {
    addRamen();
```

```
        addVeggies();
    }); // 3 minutes
    // in total, will only take 3 minutes!
```

Notice that we are passing in an *anonymous function* to `.then()`, which is executed upon completion of `boilWater()`.[20] Now, everything works as expected. We can also write this another way, which we will actually prefer throughout this workshop series.

20: When the code within a promise is completely executed, the promise is said to have been **resolved.**

2. Using `async/await`

```
    const prepareRamen = async () => {
        cutVeggies();
        await boilWater();
        addRamen();
        addVeggies();
    };
    // in total, will only take 3 minutes!
```

Looks a bit cleaner, right? The await keyword is similar to `.then()` in that we will wait for the promise returned by `boilWater()` to be resolved before we continue in our code. Note that when using `await` we must wrap our code in a function marked `async`, otherwise we will get a syntax error.

Thinking beyond this example, we typically utilize asynchronous code in any situation that involves many intensive independent tasks. For example, loading a 4k image. We don't want to halt our entire program to wait for the image to load, so instead we load it in the background using asynchronous programming. Now that all of that's out of the way, we can move on to actually using asynchronous programming in the context of MongoDB!

Mongoose Queries

Below is a snippet of code that builds upon our mongoose schema defined in the above code sample and defines an asynchronous function. The snippet below first includes the specified model, then defines an asynchronous function to return all documents matching our defined schema using the find() function. We return these documents by setting them in the result object.

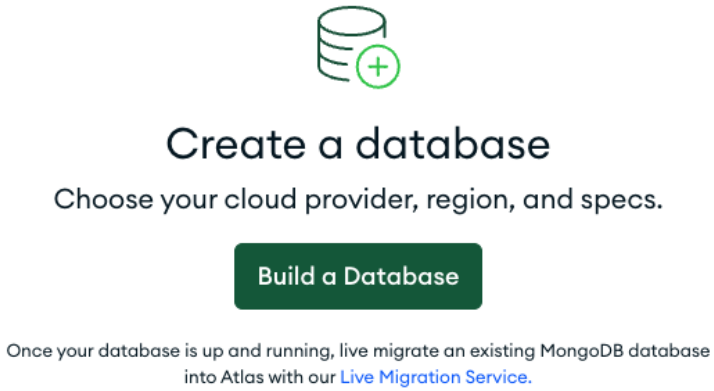
```
    const Ingredient = require("../models/Ingredient");

    module.exports.getIngredients = async (req, res) => {
        const ingredients = await Ingredient.find();
        res.send(ingredients);
    }
```

2.6 Demo

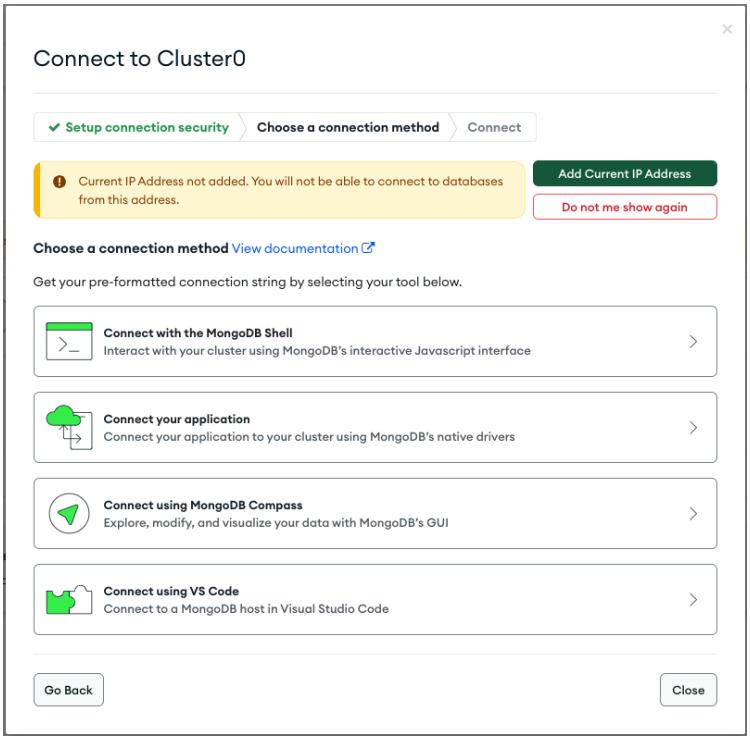
Note: This demo has not been fully fleshed out yet and may need some revamping.

Let's get started on our Twitter clone! Before we do anything else, let's set up the database (and hopefully learn something in the process). First, we will make an account with MongoDB's interactive web client: Atlas.

1. Create database. Use the free tier with all defaults. Pick a username and password and write them down. You will need them in an upcoming step.



2. You have made a "cluster."
3. Click "Browse Collections", you can view all your data here. You shouldn't have any yet.
4. Go back, click "Connect your application"



5. Create application `server.js` within backend folder using the following steps

- Make sure you have everything installed (yarn, node).
- Run `yarn init` and select all defaults.
- Copy connection string from atlas and replace password and username with the one you selected.
- Install mongoose using yarn if not already installed

6. Do the connection steps.

```
const mongoose = require('mongoose');
connection = "mongodb+srv://USERNAME:PASSWORD@cluster0.
hekc5ta.mongodb.net/?retryWrites=true&w=majority"

mongoose
    .connect(
        connection,
        {
        useNewUrlParser: true,
        useUnifiedTopology: true,
        }
    )
    .then(() => console.log('Connected to DB'))
    .catch(console.error);

```

7. Do the connection steps (replace USERNAME and PASSWORD with the proper values).

```
const mongoose = require('mongoose');
connection = "mongodb+srv://USERNAME:PASSWORD@cluster0.
hekc5ta.mongodb.net/?retryWrites=true&w=majority"

mongoose
    .connect(
        connection,
        {
        useNewUrlParser: true,
        useUnifiedTopology: true,
        }
    )
    .then(() => console.log('Connected to DB'))
    .catch(console.error);

```

8. Create a data model for our app. We will create a model for posts.

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

const Post = mongoose.model("Post", new mongoose.Schema
({
    content: {
        type: String,
        required: true
    },
    user: {
        type: String,
        required: true
    },
    num_likes: {
        type: Number,
        default: 0
    },
```

```
        timestamp: {
            type: Number
        }
    }));

    module.exports = Post;

```

9. Push data to the database.

```
    // ADD DOCUMENT
    const Post = require("./models/post"); // import Post
    data model

    const intro = new Post({ // populate required fields
    content: "Some content!",
    user: "Me",
    });

```

10. Retrieve data (we now have data persistence!)

```
    // GET DATA
    Post.find({})
        .then(posts => console.log(posts));

```

11. Modify the data.

```
    // MODIFY DATA
    Post.findById("63c5e192e6e28a4adef4cb4a")
        .then(post => {
            post.content = "Some OTHER content!"
            post.save();
        })

```

12. Neat! There are more functions to retrieve and modify data, but we will explore these as necessary.

Slides and Recording

The slides for this chapter can be found here. The recorded presentation can be found here.

Servers 3

"Simplicity, carried to an extreme, becomes elegance."

– Jon Franklin

Servers are probably one of the most misunderstood concepts for new developers. If you put ten new developers in a room, it's a pretty good bet that they've all *heard* of servers. Maybe they've been exposed to them through pop culture. They've seen movies or read books where the nerdy, basement dwelling side character is approached by the charismatic protagonist to "hack into the mainframe" to stop the evil corporation and save the world.[1] Or maybe they've come across the terminology at some point while learning about the fundamentals of programming, with their instructors glossing over it saying, "don't worry about this yet." Whatever the case may be, its likely that a majority of the ten new developers you have confined to a room would not be able to tell you what exactly a server does, or why. Or even more fundamentally, what *is* a server?

1: Note that a mainframe is just a special name for a server that is capable of performing a large amount of concurrent operations. Whether or not "hacking" into one will save the world is another question.

In a way, this lack of understanding almost serves as a hint to what a server is: a blackbox[2] to process and retrieve information. We see this concept, **abstraction**, fairly frequently in programming and Computer Science. Through abstraction, we make it far simpler for others to interact with our programs. It's a very important concept, and we'll be digging into it in detail throughout this chapter.

2: A blackbox is a term for an object that takes some input and transforms it into some desired output, with the user not necessarily knowing the details of how it works.

This write-off of servers as blackboxes is great if we just want to use them to get some data. It makes our job much easier! In fact, you interact with servers (indirectly) every single day just by browsing the internet.[3] However, when it comes time to create our own, it's important to have a deeper understanding. And that's what we aim to accomplish here! By first instilling in you an idea of the *fundamental* concept of a server,[4] and later showing one possible implementation (among many), we'll break the blackbox open and expose the ideas within.

3: Can you imagine if you had to be familiar with all the intricacies of servers just to watch a YouTube video?

4: Note that we won't go over all the low level implementation details. That's for your upper division CS classes to cover!

3.1 Servers In General

Put simply, a server is a computer like any other. What distingishes a regular old computer and a server is that **servers are given the task of listening and responding to requests**. These tend to be requests for data or to perform some task and in general they come from other computers.[5] We call these "other" computers **clients**. You can think of the interaction between a server and a client in much the same way as the interaction between a customer at a restaurant and the restaurant's staff. Just as a customer can request a glass of water, new silverware, or a half serving of Tiramisu, a client can request some function to be performed or data to be processed and returned. This brings up an important question: how do the client and server communicate? A customer at a restaurant might use English or Portuguese, but unfortunately computers aren't

5: We will see that it's not always the case that requests originate from other computers. A single computer can be both the server and the client, and you'll see that this is actually very common, particularly during the development process of a full stack application.

quite there yet. They must have some standard, agreed upon language in order to do so.

The Language of Requests

In the context of clients and servers, the "language" that is typically used is **HTTP**, or Hypertext Transfer Protocol.[6] This protocol makes it easier for servers to parse through a client's request due to the fixed format. Take a look at the following example of a real HTTP request:

```
GET / HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html,application/xhtml+xml,application/xml;
        q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
```

6: Note that there are other protocols that can be used, such as WebRTC, and each have their advantages. For now, let's not get into the weeds too much, but I recommend reading up on protocols if you're interested.

It may seem strange and hard to read as a human, but it is perfectly formatted for computers. We don't have to worry about the exact formatting as creating these requests is typically automated, but I do want to point out one key detail: the word `GET`. `GET` indicates to the server the particular action desired by the client, and it is one of several so called **HTTP request methods**. We'll discuss these in more detail and show several examples, so don't worry if you haven't quite grasped the concept yet. For now, here are a few essential methods to be aware of [7]:

- `GET`: indicates a request for some data
- `POST`: submits data to the server which often results in some side effect or change to the server's state
- `PUT`: submits data to the server in order to update an existing resource
- `DELETE`: removes some resource from the server

7: There are methods beyond these. Check out Mozilla's article on the subject if you're interested.

After receiving a well-formed request, the server will perform the specified action and create a **response** to send back to the client. The format of a response is also standardized by HTTP, and here is an example:

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 155
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close

<html>
<head>
    <title>An Example Page</title>
</head>
<body>
    <p>Hello World, this is a very simple HTML document.</p>
</body>
</html>
```

The first thing you might notice is that the response seems to have HTML embedded into it. Why might that be? Let's come back to that. Take a look at the first line of the response. As before, it indicates that it is following HTTP, but it also has the number 200 and the word `OK`. This is known as an **HTTP status code** and it represents the result of the server's attempt to address the client's request. In this case, `200 OK` indicates that the request was successfully, received, understood, and accepted. There are many response codes but they all fall into the following categories:

- `1XX`: informational; the request was received and is being processed
- `2XX`: successful; the request was successfully, received, understood, and accepted
- `3XX`: redirection; further action needs to be taken in order to complete the request
- `4XX`: client error; the request contains bad syntax or cannot be fulfilled
- `5XX`: server error; the server failed to fulfill an apparently valid request

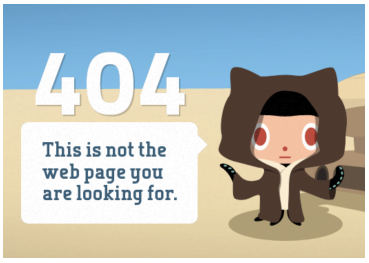


Figure 3.1: GitHub's 404 page

You don't have to memorize these, but you'll find that after working with HTTP requests for a while they'll just come naturally. For example, you might be familiar with the infamous code `404`, which indicates that a resource was not found. You'll come to recognize other codes just like this one.

About the HTML we saw before...

What was it doing there? It's known as the body of the response, and it's being sent back to the client, in essence, because that's what they asked for. HTTP messages can be broken up into two parts: the header and the body. The **header** contains metainformation about the message, while the **body** contains the data associated with the message (such as HTML or JSON). Let's break down what's going on in this particular example. The client sent a `GET` request, asking the server to send some data back from a particular location (www.example.com). The data that was sent was this HTML code... Do you see where this is going yet?

We know that HTML is used by browsers in order to render web pages, so our client can now successfully render the web page stored on the server. In essence, the client uses this HTTP request in order to receive the data necessary to render a web page! This process happens billions of times per day, and it is the back bone of the whole internet. The internet is built upon servers which store HTML, CSS, and Javascript and your browser uses HTTP requests to request them to be sent to you! Obviously, there's more to the internet than just this,* and we could fill many books talking about it, but it's outside the scope of this workshop series. If you're interested take CS 118!

*This section in particular is called the Web.

HTML is not the only thing that can be placed in response bodies. In fact, just looking at the `Accept` section of our HTTP request we can see that images can be as well!

```
Accept: text/html,application/xhtml+xml,application/xml;
```

```
q=0.9,image/avif,image/webp,*/*;q=0.8
```

Another common data format used in HTTP bodies is known as **JSON**, or Javascript Object Notation. We'll discuss JSON more in detail once we actually see it in action, but for now it suffices to understand that it is a way to encode objects in Javascript as strings. For example, the following code block shows an object called `heck` and its corresponding JSON string representation:

```
heck = {
    studentOrgRanking: 1,
    color: "#C960FF",
    rizz: 100,
    website: "https://hack.uclaacm.com"
}

{
    "studentOrgRanking": 1,
    "color": "#C960FF",
    "rizz": 100,
    "website": "https://hack.uclaacm.com"
}
```

Figure 3.2: Rizz 100

3.2 Web APIs: What's on the Menu?

Now that our server and client have a common language, it's time to take things a step further. Let's revisit the restaurant analogy. How does the customer know what they're allowed to order? They can't just demand to be served whatever they want, because the restaurant might not be able to accommodate their request.[8] That's why every restaurant has a menu! There needs to be a way to let customers know what they can order. Clients and servers are much the same. There needs to be an understanding between them about what the server can do for the client, and this is accomplished using the **API**, or Application Programming Interface. You may have heard this term before. It's another one of those nebulous phrases that gets thrown around a lot, but is rarely defined concretely.

8: Yes, there are exceptions, secret menus, etc. But let's be real. If you order off the secret menu, the staff hates you.

In general, an API is just a way for two computer programs to interact with each other. Think of the customer at a restaurant as one program and the staff as another. The customer hasn't eaten in 16 hours and is craving a burrito with carnitas and guacamole.[9] Using the menu (the API), the customer is able to enjoy the result of the staff's work and they don't need to attend 4 years of culinary school in order to do it! Put another way, the API allows us to interact with a blackbox and receive meaningful results. APIs can be found everywhere in software engineering, but we will be creating more specialized APIs called **Web APIs**.[10]

9: I am so hungry right now.

10: As the name implies, these are APIs that utilize the Web, allowing communication from client to server through HTTP requests.

Let's take a look at an example of a simple API.

API's can be expressed in several ways, either using code or English. Let's keep it simple and just use English. Consider a server with the sole purpose being to simulate a cat. The API defines several actions

that you, as a pet owner, can take to interact with the cat, and in response the server will send a JSON string (recall that JSON is just a string representation of a Javascript object) with information about the cat and its actions.

The API is defined as follows:

```
POST /FEED: You feed the cat.
POST /WATER: You give the cat a drink.
POST /PET: You pet the cat.
GET /STATUS: You check how the cat is doing.
POST /MEOW: You meow at the cat.
```

These five actions define how you can interact with the cat server. Some of the interactions may have side effects, or an effect on the state of the cat. Also, notice the HTTP method names before each action name! Let's start by petting the cat. Note that the formatting below does not follow HTTP.

```
REQUEST: POST /PET

RESPONSE:
{
    "health": 100,
    "hunger": 10,
    "thirst": 10,
    "action": "Meows and sits down, ready
                for more pets."
}
```

He seems friendly! Let's give him some food.

```
REQUEST: POST /FEED

RESPONSE:
{
    "health": 100,
    "hunger": 50,
    "thirst": 10,
    "action": "Meows gratefully,
            and attacks the food."
}
```

Okay, he seems to be enjoying that. Let's pet some more.

```
REQUEST: POST /PET

RESPONSE:
{
    "health": 100,
    "hunger": 50,
    "thirst": 10,
```

```
        "action": "Bites your hand. He wasn't
            done eating yet!"
    }
```

Ouch. How to respond?

```
    REQUEST: POST /MEOW

    RESPONSE:
    {
        "health": 100,
        "hunger": 50,
        "thirst": 10,
        "action": "Looks up from food, confused."
    }
```

Okay, that's enough playing with the cat! Hopefully, this toy example gave you a clearer idea of what an API is as well as its purpose. We'll be creating a real API later in this chapter using Javascript.

Typically, Web API's contain multiple **endpoints**. In general, an endpoint can be thought of as a point of contact between a client and a server. Depending on which endpoint is invoked, the server knows which action to take. In the previous example, we can think of each of the possible five actions as an endpoint. Another common way of thinking about endpoints is as *specific digital locations* of resources located on a server. For example, if we want to access the resource located at `/STATUS` on a server, we use the `GET /STATUS` endpoint. You can think about endpoints in whichever way is best for your own mental model, as long as you remember that endpoints are meant to direct the server towards a particular action or resource. And don't worry if things aren't clear yet! We'll be showing concrete examples of all of these concepts in the next few sections.

Figure 3.3: This is the cat. His name is Kevin.

3.3 Server Implementations

As one might expect, there are many ways to go about implementing a server. We know that a server is simply a computer tasked with listening and responding to requests, and clearly this task is not specific to any single programming language. Some popular choices are the following:

- Node.js and Express
- Python and Django
- Python and Flask
- C and Pain[11]

There are countless libraries out there, so no need to reinvent the wheel. Since we're focusing on the MERN stack for Stackschool, we'll be going with Express and Node.[12] To avoid any confusion let's first discuss what exactly they do, and why they are useful for us when building a server app.

11: Pain in the literal sense of the word. Not recommended.

12: Representing the E and N in MERN respectively.

What is Node?

Originally, Javascript was created as a scripting language for the browser, Netscape, and wasn't intended to be executed outside of that environment.[13] However, as time has gone on and Javascript has gotten more popular, it has transcended this original functionality. In 2009, a man named Ryan Dahl decided that Javascript would be an excellent language for writing server programs, and created a new runtime environment[14] for it outside of the browser. Node was the product of his efforts. Now, years later, it's the most popular non-browser runtime for Javascript and home to a thriving, community driven ecosystem of libraries.[15] As described on their website, Node is "an asynchronous event-driven JavaScript runtime designed to build scalable network applications." Essentially, it's perfect for servers!

13: It also took only 10 days for the first version to be developed, which honestly explains a lot.

14: By runtime environment, I just mean a program that can execute code.

15: We call these libraries "packages", and we access them using a program called NPM, or Node Package Manager. Another popular (and, in my opinion, better) package manager is Yarn. We'll be using Yarn for all examples here.

What is Express?

Node gives you all the fundamentals required to make a server, but why do that if someone's already done most of the work for us? Express is a framework that makes creating server applications far easier by abstracting away most of the HTTP request logic. There are plenty of alternative frameworks, but to stay true to the MERN stack, we'll be going with Express.

Creating Your First Server

Now that we have all that out of the way, let's get to the fun part. First, make sure you have all the necessary installations. Follow the checklist:

- Node. If you don't already have it installed, here is an extensive guide to doing so no matter which platform you're on. I recommend using Homebrew if you're on MacOS.[16]
- Yarn. This will be your package manager, or the program you use to install node packages (like Express). Once you have Node installed, you can install Yarn by running `npm install --global yarn` in your shell.

16: Some would recommend using a version manager like NVM, but if you just want to get your hands dirty quickly, the other methods are adequate for now.

Now in your terminal, navigate to the directory of your choice[17] and run `yarn init`. This will start the process of creating your server application. It'll prompt you with some basic configuration information, but you can just accept the default values by pressing enter for each one. Don't worry, you can change these values later! After you've completed this step, a new file called `package.json` should have been generated. This is the configuration file for your server. To install Express, run `yarn add express`. This should generate a directory called `node_modules` and a file called `yarn.lock`.

17: If you're not familiar with navigating the terminal, check out this resource.

Now let's make the server itself. Create a new file called `server.js`. Within this file, add this boilerplate code:

```
1  /**** INIT SERVER ****/
2  const express = require('express');
3  const app = express();
4  app.use(express.json());
```

```
5
6       /**** DEPLOY SERVER ****/
7       const port = 8080;
8
9       app.listen(port);
10      console.log(`listening on http://localhost:${port}/`);
11      console.log("Press Ctrl-C to quit");
```

Congrats, you just made your first server! Unfortunately, it doesn't do anything. You can run it with `node server.js`. Before making it a bit more useful, let's break down what exactly is happening. Take a look at the second line. In Node, the `require` function is a way to include code from other files within your file.[18] In this case, we're including code from the Express package. In the next line, we create an Express app and bind it to a variable. In the final stage of initialization, we tell the app to use something called a **middleware** function. We'll get into these in more detail soon, but for now just know that it's a way to make your life easier. Now, in the deployment section, we define a port, and tell the server to listen on that port.[19] This will affect the URL of your local server.

Alright, now that we've cleared all that up,[20] its time to add our first endpoint. We'll make an endpoint that requests a random number from the server. It's pretty easy!

```
6       /**** ROUTES ****/
7       app.get("/random", (request, response) => {
8           // generate random number from 1-100
9           const rand = Math.floor(Math.random() * 100) + 1;
10
11          // send random number in response
12          response.send(`${rand}`);
13      })
```

This endpoint can be referred to as `GET /random`[21] and every time it is invoked it will return a string containing a random number from 1-100. You can test it out by starting the server and visiting `http://localhost:8080/random` in your browser.[22] At a surface level, all that's going on here is that we're using Javascript code to define our server's API.

Describing the API with English

Recall how we defined our API in the cat server example. We can describe our Javascript endpoint definition for random in the same way!

```
GET /random: Get a random number from 1-100.
```

They're two ways of saying the same thing, except that one of them happens to be real, functional code.

Hopefully, you now have a feel for the general process of creating Express applications. You're now ready to put everything together into a more realistic application.

18: Similar to imports/includes in other languages you may be familiar with. We call the code that we're importing a "module."

19: Ports allow you to host multiple servers on the same machine, each on a different port.

20: May be worth a couple more readthroughs if it's still unclear, or get in contact with us on Discord!

21: Note that `/random` is known as a **route**. The distinction between endpoints and routes is that endpoints include the HTTP method (i.e. GET, POST, etc.) in their definition. You can have multiple endpoints with the same route, as long as the method is different (so having `GET /random` and `POST /random` would be perfectly fine).

22: In order to start your server, use `node server.js`. Also, if you're wondering what the deal with localhost is, know that it's a special domain name that represents the current computer.

3.4 Demo

First, let's get the express app set up (read: boilerplate). Remove the code from last time starting at line 17, we won't be needing it anymore. Add the following to the top of the file (and ensure you have express installed on your project[23])

23: If not, `yarn add express`.

```
1  const express = require('express');
```

And add the following to the bottom of the file:

```
1  const app = express();
2  app.use(express.json());
3
4  app.listen(3001, () => console.log('Server listening on port
   3001'));
```

We explained what's going on here in section 3.3 so check that out for a reminder! At this point, your file should look like this:[24]

24: With the exception of the string having your actual username and password.

```
1  st express = require('express');
2  const mongoose = require('mongoose');
3  mongoose.set('strictQuery', false);
4
5  const app = express();
6  app.use(express.json());
7
8  // INIT CONNECTION
9  mongoose
10     .connect(
11         "mongodb+srv://USERNAME:PASSWORD@cluster0.hekc5ta.
   mongodb.net/?retryWrites=true&w=majority",
12         {
13         useNewUrlParser: true,
14         useUnifiedTopology: true,
15         }
16     )
17     .then(() => console.log('Connected to DB'))
18     .catch(console.error);
19
20 const Post = require('./models/post');
21
22 app.listen(3001, () => console.log('Server listening on port
   3001'));
```

Now you can try running it to make sure it works with `node server.js`. Alright, boilerplate is out of the way now. Let's get to the interesting part! Let's think about what we might want to include in the backend API of a twitter clone. Recall that last time we set up a data model for our posts. What should we have in our API related to posts? Well, first of all, we have to actually get the feed of posts, so one endpoint related to getting a list of posts would be helpful. We should also be able to create new posts. If we in retrospect decide a post doesn't reflect on ourselves as well as we would have hoped, we can edit it or simply delete it all together. Finally, if we see a post we like, we want to be able to like it. So all together, we should create the following endpoints:

- getting the feed
- creating new posts

- editing posts
- deleting posts
- liking posts

So let's do that! First up, we'll create the `GET /feed` endpoint. Recall our discussion in the last chapter about asynchronous programming and mongoose. These concepts will be essential here![25]

25: And I recommend going back and rereading if you need a refresher.

```
1    // Get posts feed
2    app.get('/feed', async (req, res) => {
3        const feed = await Post.find();
4
5        res.json(feed);
6    });
```

This code indicates to our server that it should listen for `GET` requests on the `/feed` endpoint. Once it "hears" something, it will run the anonymous asynchronous function. In this case, it will retrieve all documents from our database that conform to the `Post` data model, and then return a json representation of them in the response object. Notice that we must `await` the result of `Post.find()` since it is an asynchronous function (and our response is dependent on its result).

Now we want allow for the creation of new posts. To do this, we'll create a POST[26] endpoint at `/feed/new`. First, however, we need to discuss how to pass data in our requests. There are several ways:

26: Recall that POST requests typically update the state of the server in some way.

1. Pass data in request body
2. Pass data as URL path parameters
3. Pass data in the "*query string*"

We'll address two of these as we use them in the endpoints to come. First up: request body.

Request Body

Recall the anatomy of an HTTP request: each one includes both a head and a body.[27] Just like responses can pass data (such as HTML) in their bodies, requests can pass data in their bodies as well! As you might expect, we do this in JSON format. Here's an example of a properly formatted HTTP POST request that utilizes the body:

27: See section 3.1 for more information.

```
POST /feed/new HTTP/1.1
Host: localhost:3001
Content-Type: text/html; charset=utf-8

{
    "content": "To be honest, I've never written a raw HTTP Post request before
    and I hope I never have to again.",
    "user": "eener"
}
```

This is a request directed to the `POST /feed/new` endpoint, whose javascript specification follows below:

```
// Create new post
app.post('/feed/new', (req, res) => {
    const post = new Post({
        content: req.body.content,
        user: req.body.user,
        timestamp: Date.now(),
    });

    post.save();

    res.json(post);
});
```

Notice how we extract information from our request body using `req.body`. This is simply a Javascript object so we can access fields within it like any other. Recall from last time `post.save()` simply saves a document to our database. Finally, we close our function with a response containing a JSON representation of our post for good measure.[28]

28: Note that `.save()` is an asynchronous function, however we do not `await` its resolution in this case because our response does not depend on its return value.

Now let's create an endpoint to edit our posts. First, let's think about this. How do we identify which post we want to edit? To address this, it's important to understand that each document in our database has an associated field called `_id` which uniquely identifies it. We could send this id in the request body, but it's better style to do it as part of our URL path so let's try that instead!

URL Path

To indicate an HTTP URL path variable, we simply add a colon before it. We can then reference it using the given name. This is probably easier to understand if given an example, so lets dive in. Our edit endpoint is as follows:

```
// Edit post content
app.put('/feed/edit/:_id', async (req, res) => {
    const post = await Post.findById(req.params._id);

    post.content = req.body.content;
    post.save();

    res.json(post);
});
```

Everything here is pretty self explanatory if you've been following so far, but I do want to point out that we access our URL parameter using the `req.params` object. We then pass in the new content we want for our post in the request body. Also note that we use a `PUT` request here, as we are updating an existing resource.[29]

29: We could also use a `POST` request, but best practice is `PUT`.

We'll speed through the rest of our post endpoints.

```
// Delete post
app.delete('/feed/delete/:_id', async (req, res) => {
  const result = await Post.findByIdAndDelete(req.params._id);

  res.json(result);
});

```

```
    // Like post
    app.put('/feed/like/:_id', async (req, res) => {
      const post = await Post.findById(req.params._id);

      post.num_likes++;
      post.save();

      res.json(post);
    });
```

What else might we need to consider when making our app? Another major component is users. We should probably make a data model for that!

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

    const User = mongoose.model("User", new mongoose.Schema({
        username: {
            type: String,
            required: true,
          },
          password: {
            type: String,
            required: true,
          },
    }));

    module.exports = User;
```

Don't forget to import this data model into your `server.js` file. This is all we need for now, but we'll definitely be coming back to this.[30]

30: We need to protect our users data! Storing passwords in plain text is no good.

Now let's think about what endpoints we should add related to users. Similar to posts, we should have a way to get a list of users, create a new user, delete a user, and edit a user's information. We won't explain these as they are fairly similar to what we did for posts.

```
    // Get all users
    app.get('/users', async (req, res) => {
      const users = await User.find();

      res.json(users);
    });

    // Create new user
    app.post('/users/new', async (req, res) => {
      const user = new User({
        username: req.body.username,
        password: req.body.password,
      });

      await user.save();

      res.json(user);
    });

    // Delete user
    app.delete('/users/delete/:_id', async (req, res) => {
      const result = await User.findByIdAndDelete(req.params._id);
```

```
23
24      res.json(result);
25    });
26
27    // Edit user information
28    app.put('/users/edit/:_id', async (req, res) => {
29      const user = await User.findById(req.params._id);
30
31      user.username = req.body.username;
32      user.password = req.body.password;
33      user.save();
34
35      res.json(user);
36    });
37    ```
38
39    We also need a way for users to log in. This one is *slightly*
       more complicated, but not too bad yet...
40
41    ```js
42    // Log in user account
43    app.post('/login', async (req, res) => {
44      const user = await User.findOne({ username: req.body.
     username });
45      if (!user) {
46        res.json({ 'error': 'That username doesn\'t exist'})
47        return;
48      }
49
50      if (user.password === req.body.password) {
51        res.json(user);
52      }
53      else {
54        res.json({ 'error': 'Incorrect password'})
55      }
56    });
```

Note that with this logic, we should also refine our user creation endpoint to ensure there are no duplicate usernames (otherwise it would be trivial to hack into somebodies account). The following modification should do the trick:

```
1     // Create new user
2     app.post('/users/new', async (req, res) => {
3       const dupUser = await User.findOne({ username: req.body.
     username });
4       if (dupUser) {
5         res.json({ 'error' : 'Duplicate username exists.'})
6         return;
7       }
8       const user = new User({
9         username: req.body.username,
10        password: req.body.password,
11      });
12
13      await user.save();
14
15      res.json(user);
```

With that, congratulations! You've successfully implemented your very own backend application using Express! Hopefully there aren't any bugs...

3.5 Testing

Speaking of, how can we actually test our endpoints? After all, we want to ensure everything works as expected. For `GET` requests, it's as simple as copying a link into your browswer. If we start our server and visit `http://localhost:3001/feed`, we should see a JSON representation of our posts. However, what if we want to send a request with a body? How can we go about doing that? There are several ways:

1. Use a GUI application like Postman – Postman makes it easy to form request bodies. Unfortunately, you have to download a whole new app.
2. Use a requests library like axios – This seems like a good idea, and in fact we will be doing it later in this workshop series!
3. Use a VSCode extension – Less intuitive than Postman, but at least we don't have to download another app! Let's dive in.

The extension is called REST Client, and you can download it like any other VSCode extension. Then in order to use it, create a new file ending in `.http`. We're going to be writing raw HTTP requests, but don't worry! You can simply copy paste the boilerplate from below:

```
[METHOD] [/endpoint] HTTP/1.1
Host: localhost:3001
Content-Type: application/json; charset=utf-8

[body]
```

The [METHOD] field indicates which HTTP request method we are using i.e. `GET`, `POST`, etc. [/endpoint] is our endpoint path and [body] is our body. Note that our Content-Type here is `application/json`, so we expect to see JSON in our body (if anything). To test an endpoint, start your server, fill in the fields with the information of your choosing, and hit "Send Request" above the request head. Try testing out all the endpoints we created!

3.6 Organization

To be added. For now, read this.

3.7 Documenting Your API

To be added. For now, read this.

Slides and Recording

The slides for this chapter can be found here. The recorded presentation can be found here.

Backend Integration | 4

"Coming together is the beginning. Keeping together is progress. Working together is success."

Henry Ford

Now that we're backend experts (more or less), it's time to shift gears. In this chapter we'll be focusing on building the frontend and integrating our backend into it. This will mostly involve a lot of review of frontend topics, although we will be covering several new ideas you've likely never seen before. In particular, how do we make calls to our backend from our frontend? We'll explore this, and more, in the chapter to come.

As mentioned in the Introduction chapter, it will be useful to have some basic background knowledge on several technologies going into this chapter. This includes:

- Fundamental Coding Concepts (Think CS31)
- Basic Shell Commands (`cd`, `ls`, etc.)
- HTML/CSS
- Javascript
- React

I recommend checking out our past workshop series, Hackschool, for a refresher on these. Without further ado, let's get started.

4.1 The Feed

It seems most natural to start with the feed for our twitter clone, so let's do that. As good software developers, let's try to brainstorm some things we might need to do before we jump in.[1] First of all, we know that all of our posts are stored in our MongoDB database. We can use one of our endpoints to retrieve them![2] This brings up an important question, however: how can we programmatically make HTTP requests? Recall that in the last chapter we saw there are several ways we can make HTTP requests. As a refresher, here they are again:

1. Use a GUI application like Postman
2. Use a requests library like axios
3. Use a VSCode extension

It seems that options 1 and 3 are moreso meant for testing rather than any programmatic use, so we're left with one option! Axios is a promise-based[3] HTTP requests library that abstracts away many of the tedious details involved with making HTTP requests. Rather than tell you what it can do, let's just jump in and show an example of a function that utilizes it. Going off of our feed motivation from before, let's write a function that invokes the `GET /feed` endpoint from our backend and add it to our

1: And make a plan to address each one.

2: Specifically, `GET /feed`.

3: That means asynchronous!

React app. Make sure you run `yarn add axios` in your frontend project. To test it out, start your backend in another terminal.

```
import axios from 'axios';
const URL = "http://localhost:8080";

//Gets the entire feed
function getFeed() {
    axios.get(URL + "/feed")
        .then(response => {
            console.log(response.data);
        })
        .catch(console.error)
}
```

You should see a log in the console containing your feed! Congratulations, you have officially taken the first true step towards making a full stack application. Recall that axios uses promises, so we must incorporate one of the promise resolution methods we discussed in chapter 2 (in this case `.then()`).

If we try calling this function, we see the Array of posts in our MongoDB database in the console. Pretty good! We can now try displaying it on our frontend.

Mapping

Before we go any further, let's think about what we want to accomplish here. We want to somehow iterate through each post in our posts array, and display information from each one. Up to this point, the canonical way you have been taught to do this is to use a loop! It might look something like this:

```
for (let i = 0; i < posts.length; i++>) {
    // display post info for current index
}
```

However, there are a couple problems with this. The biggest among these is that we need to write our code within a JSX return block, which we aren't allowed to do! The `return` needs a value following it, so we can't just add a for loop after it. To fix this, we use mapping!

Mapping is a way of iterating over an array and performing a set of operations on each item within it. Once we complete iteration, it returns a new array with the operations performed! It can be thought of as a kind of "transformer" function. It transforms the items of an array into a new format using some function and spits out the result. Here's a toy example:

```
a = ["Nathan", "James", "Nareh", "Christina"]; // array to
iterate over

pog = (name) => { // operation to perform
    return `pog${name}`;
}

a.map(pog); // performing pog on each item in a
```

```
// Output: ['pogNathan', 'pogJames', 'pogNareh', 'pogChristina
']
```

We can also simplify this a bit more by utilizing anonymous functions. Rather than name our operation `pog`, let's just pass it in directly.[4]

4: We can omit our `return` keyword here using a special syntactic sugar built into JS! For this to work, we must have a single expression in our function and omit our brackets as well.

```
a = ["Nathan", "James", "Nareh", "Christina"]; // array to
iterate over

a.map((name) => `pog${name}`); // performing pog on each item
in a

// Output: ['pogNathan', 'pogJames', 'pogNareh', 'pogChristina
']
```

Cool! Let's apply this to our feed. In this case, we want to map every post object in our posts array to a JSX component! Let's do that! Recall that a post contains some content, a user, a like count, and a time stamp indicating when it was posted. [5]

5: You have noticed a warning in the console about adding a "unique key prop." To silence this, simply add `key=i` to the `div` tag.

```
return (
    <div>
    {posts.map(post =>
        <div>
        <h3> {post.user} </h3>
        <p> {post.content} - Time: {post.timestamp} - Likes: {
post.num_likes}</p>
        </div>
    )}
    </div>
);
```

Unfortunately, this doesn't seem to be working? What's going on?

React Hooks Recap

React is lazy. It always strives to do the bare minimum to display the user interface. In particular, it only wants to refresh the UI when there is a visual change to be made. In order to enable this laziness, some smart people designed the `useState()` hook.[6] Using this hook, we can designate a variable to be "watched" for changes. If it changes, then the UI will refresh. Sounds cool right?

6: Recall that a hook in React is just a function that typically starts with "use" and performs some component related logic (usually).

In general, `useState()` looks like this:

```
const [watchedVar, setWatchedVar] = useState([DEFAULT_VAL]);
```

The syntax seems a bit funky, but all thats going on is that useState returns an array of two items: a variable to be watched and a function to set the watched variable. We pass in a default value to the function and we call the setter function when we want to update variables value. Seems a bit convoluted, but trust me when I tell you it more than makes up for it in practice. For a more detailed explanation of `useState` check out one of our previous workshops!

Consider our feed example. We want our UI to update when we fetch our posts array from our server. We can use `useState()` as follows:[7]

7: Don't forget to import `useState`.

```
const [posts, setPosts] = useState([]);
```

By default, we'll just set posts to be an empty array. In order to set the value of posts, we call the `setPosts()` function. In fact, let's do this within `getFeed()`.

```
//Gets the entire feed
function getFeed() {
axios.get(URL + "/feed")
    .then(response => {
        setPosts(response.data); // <-- We changed this line
    })
    .catch(console.error)
}
```

As an additional caveat, we only want our API call to be performed when our app is initially loaded.[8] To accomplish this, we use `useEffect()`! In general, `useEffect()` looks like the following:

8: Currently it is being called an obscene amount every time we run our app.

```
useEffect(() => {
    // perform some task
}, [dependency1, dependency2, ...])
```

The function passed to the `useEffect()` hook will be called whenever one of the dependencies in the dependency array is updated! If the array is empty, it will only be called upon initial app load. Again, check out the workshop we linked before for more information on this, but here's how we use it in our demo app:[9]

9: Note that, just like `useState`, `useEffect` needs to be imported from react.

```
useEffect(() => {
    getFeed();
}, []);
```

We now have our feed! Granted, it looks quite ugly, but let's take this W for now.

4.2 Profiles and Navigation

4.3 Finishing Touches