Stackschool

ACM Hack

Contents

1 Introduction to Full Stack

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

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, one may reasonably question whether there is a pressing need to store data externally for simple web applications.

To fully address this concern, 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. 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. In this scenario the restaurant dining room represents our web application, the fridge represents our **database** and the eggs represents our **data**. From this example, we note that creating a shared supply of data for many users necessitates an external database to keep things consistent.

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 users 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. Again in this scenario the dining room of the restaurant is akin to a web application, while the kitchen fridge represents our database and ham represents our data.

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.

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.

2.1 Spreadsheets and Schemas

The simplest version of a database is an unstructured file. Assuming our web application has read and write access to our file[2], we 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. Instead we will begin by considering its cousin, the **spreadsheet**. A spreadsheet is a file with columns and rows - effectively a large **table**. 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

2: Giving a program read access to a file allows it to open the file and access its contents. Giving a program write access allows the program to modify the contents of a file.

types are the comma separated file format (.csv), and Microsoft Excel's various formats (.xls, .xlsx, .xlsm).

Let's modify our restaurant scenario slightly to align more closely with a traditional database. 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. **Figure 2.2** displays a simple example of such a note. The set of column headers 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.

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 quantity.

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.

	A	B	C
1	Item	Quantity	Price
2	Lettuce	20	3
3	Tomatoes	20	3
4	Broccoli	30	4
5	Carrots	80	3

Figure 2.1: A simple spreadsheet made with Microsoft Excel

	A	B	C	D	E
1	Item	Quantity	Price	Last_Updated	Update_By
2	Lettuce	20	3	12/25/2022	Joe
3	Tomatoes	20	3	1/3/2023	Bob
4	Broccoli	30	4	1/5/2023	Fred
5	Carrots	80	3	12/20/2022	John

Figure 2.2: A spreadsheet with metadata fields

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 persist 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[3]. Another key feature shared by most popular relational databases is their use of **Structured Query Language (SQL)** to interface with the database.

3: 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. Assuming our table in our database is named "Fridge", Below is an example query that

would generate the table in Figure 2.2

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

To create the table depicted in Figure 2.3, we would simply need to specify the additional columns present - or make use of the wildcard operator (*)

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

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" keyword, 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 **var**iable (length) **char**acter field, and must be quoted with single quotes.

```
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(),
    'ACMhack'
)
```

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[4], 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

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

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

2.3 Non-Relational Databases

An alternative (and a better choice in certain scenarios) to a relational database is a **non-relational database**, frequently referred to as a **NoSQL database**[5]. Like the name implies, non-relational database are structured differently from their relational counterparts.

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

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 **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 2.4.

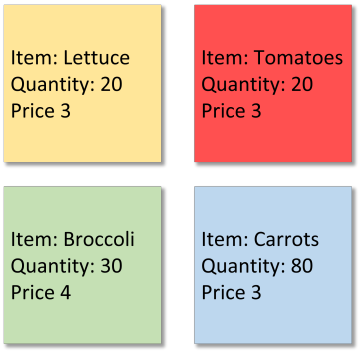

Figure 2.3: A visual representation of a non-relational database

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 an 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 use 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[6] 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**.

6: 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 we'll be using in our MERN[7] tech stack) 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

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

code that would represent each of the four documents in our restaurant example.

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

    tomatoes = {
        _id: 10000123456879,
        __v: 0,
        item: "Tomatoes",
        quantity: 20,
        price: 3,
    }

    broccoli = {
        _id: 20000123456879,
        __v: 0,
        item: "Broccoli",
        quantity: 30,
        price: 4,
    }

    carrots = {
        _id: 30000123456879,
        __v: 0,
        item: "Carrots",
        quantity: 80,
        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 most 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[8])
- Programmability: can define stored procedures and functions to streamline queries

8: 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 database 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 an a dedicated computer you can guarantee will be continuously running. After configuring the connection between your web application and database computer[9], 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.

9: 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, after consulting the notes on the fridge, let's assume the chef must put on some slippers before entering the fridge to acquire the relevant ingredients and cook the desired meal. In our MERN tech stack, these slippers are akin to the **mongoose** library, a JavaScript Object Data Modeling library for MongoDB that simplifies the process of retrieving and modifying data in our datanbase.

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);
```

Mongoose also offers a wide variety of intuitive functions manipulate and add new documents to the collection with a particular schema[10]. 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()
- Model.findByIdAndDelete()
- Model.findByIdAndUpdate()
- Model.find()
- Model.create()
- Model.count()
- Model.exists()

10: 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 illustrate using Mongoose's functions, we must first address a final issue we will encounter with using our database. Returning to our restaurant analogy, we can realize that the entire process of the chef acquiring the relevant ingredients and preparing a meal will take a fair amount of time. Likewise, sending and retrieving data from our application and database may also not run instantaneously[11].

11: Communicating between two distinct computers often happens over the internet a process that runs orders of magnitude slower than just running something on a single computer.

We often choose to inform our code of this possible long wait and let our code run lines that are independent of the result in the mean time. This technique of running multiple tasks concurrently is known as **asynchronous programming**. When running JavaScript code with Node.js, we must explicitly specify a functions as asynchronous with the "async" keyword. For any operation that may pause our asynchronous function, we use the "await" keyword to signify this function should be paused until the specified line has completed. In the mean time other code not in our asynchronous function may be run.

Below is a snippet of code that builds upon our mongoose schema defined in the above code sample and defines and 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.

```
1    const Ingredient = require("../models/Ingredient");
2
3    module.exports.getIngredients = async (req, res) => {
4        const ingredients = await Ingredient.find();
5        res.send(ingredients);
6    }
```

2.6 Demo

2.7 Testing

a

2.8 Organization

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

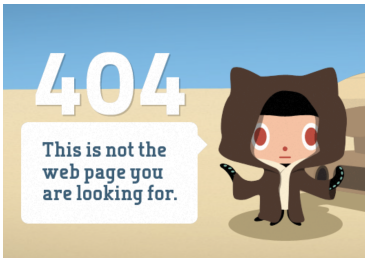

Figure 3.1: GitHub's 404 page

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 API's: 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. API's can be found everywhere in software engineering, but we will be creating more specialized API's called **Web API's**.[10]

9: I am so hungry right now.

10: As the name implies, these are API's 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

3.5 Testing

3.6 Organization

3.7 Documenting Your API

Backend Integration 4

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5 Advanced Frontend

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.