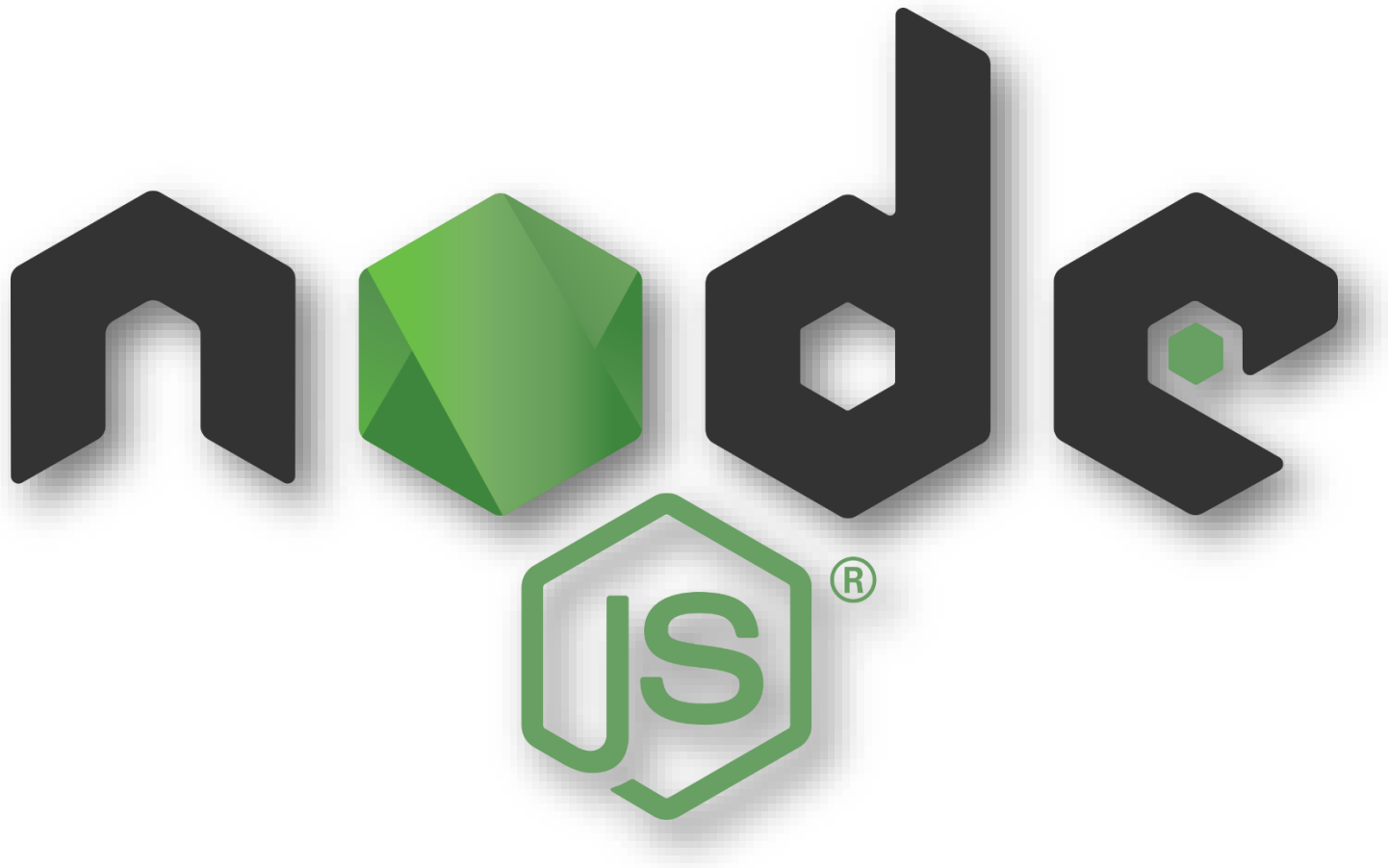
Chapter 8

Node.js & NPM

What is in This Chapter ?

This chapter begins by comparing (at a basic level) the **Node.js** architecture with other server architectures. The **event loop** and **event queue** lie at the heart of the innerworkings of **Node.js**, so they are explained as well. We will look at the **module system** of **Node.js** and make use of a few built-in ones, as well as see how we can make our code available to be used as a module as well. We will **create a few simple servers** and see how we can set up a server for our **FutureTechCorp** website. We will follow this with a discussion of how to handle incoming parameters and form data from **GET** and **POST** requests. The last section gives an introduction of the **Node Package Manager** (**NPM**) and how it can be used.



8.1 Node.js **Architecture**

Node.js is an open-source, cross-platform **JavaScript** runtime environment that allows developers to run **JavaScript** code outside of a web browser. It is commonly used for the creation of web servers and real-time applications such as chat apps and multiplayer games.

What can Node.js do?

- Generate **dynamic page content** using templating engines or frameworks
- Create, read, write, update, and delete **files on the server**
- Handle **form submissions** and parse user input
- **Interact with databases** to perform create/read/update/delete operations
- Handle thousands of **simultaneous client connections** efficiently
- many more things …

How does Node.js compare with traditional servers?

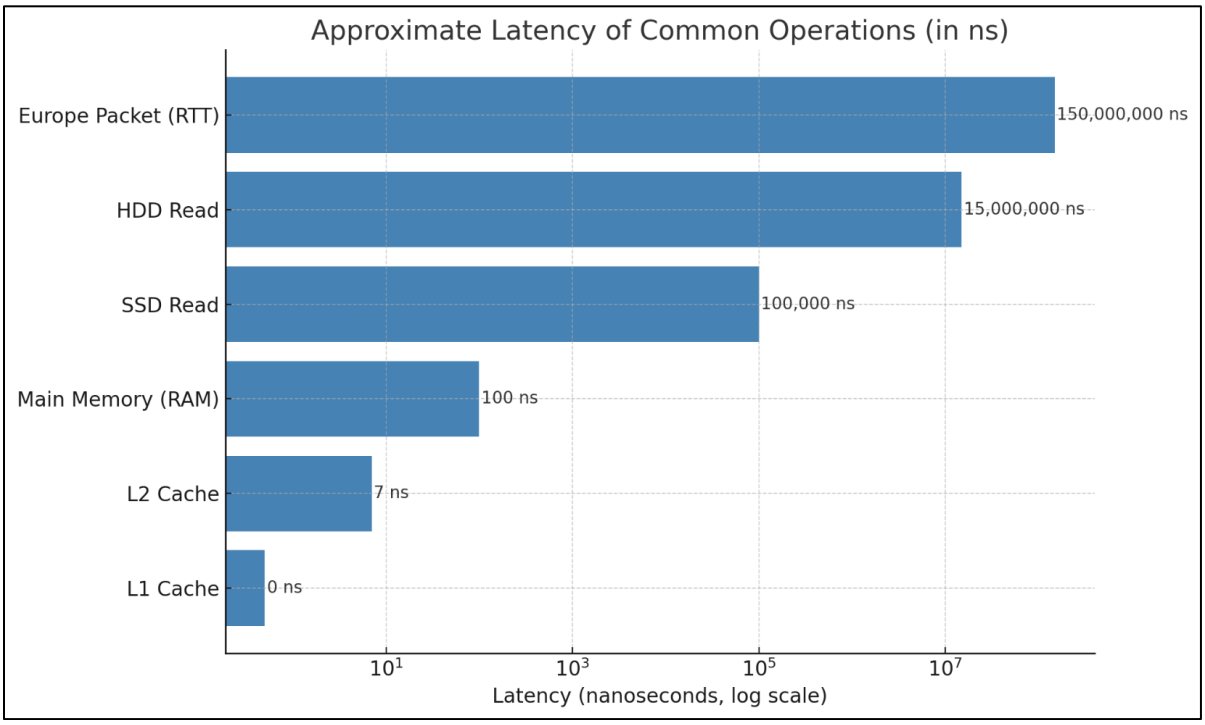
With traditional servers, when a client request comes in, it blocks the thread until it's finished. This can be slow under high loads, since threads are tied up waiting. However, with **Node.js** servers, it uses non-blocking I/O and an event loop to stay responsive. This means that it is more scalable because it handles many simultaneous requests efficiently with fewer resources.

For example, consider the difference between the two when a client requests a file:

Step	Traditional Server (Synchronous)	Node.js Server (Asynchronous)
1	Receives file request from client	Receives file request from client
2	Sends a command to file system to read file	Sends a command to file system to read file
3	**Waits/blocks** until file system finishes reading	**Continues processing other requests** without waiting
4	Once file is read, sends content back to client	When file read is done, a **callback/event** sends file to client
5	Becomes available to handle next request	Always available to handle more requests during file read process

As we can see from the above table, step 3 is where the crucial difference lies. Traditional servers block to access the file (or database, etc..) so that the server cannot handle any other incoming requests until that file is read in and returned to the client. **Node.js**, on the other hand, does not block. Instead, it assigns an event listener to handle the file reading and file sending (i.e., back to the client). In the meantime … it is able to take on additional client requests. Although this example discusses file access, the same principle applies to any operation that can be delayed.

What is the big deal about file access? Well, when dealing with I/O operations, we can measure things in terms of **milliseconds** (one thousandth of a second), **microseconds** (one millionth of a second) and even **nanoseconds** (one billionth of a second). When doing time delay comparisons, we use a term called ***latency*** … which is the **time delay** between when a request is made and when the response begins. Consider the latency differences for accessing cache memory, main memory, SSD & HDD disk memory and internet packet sending in this chart:

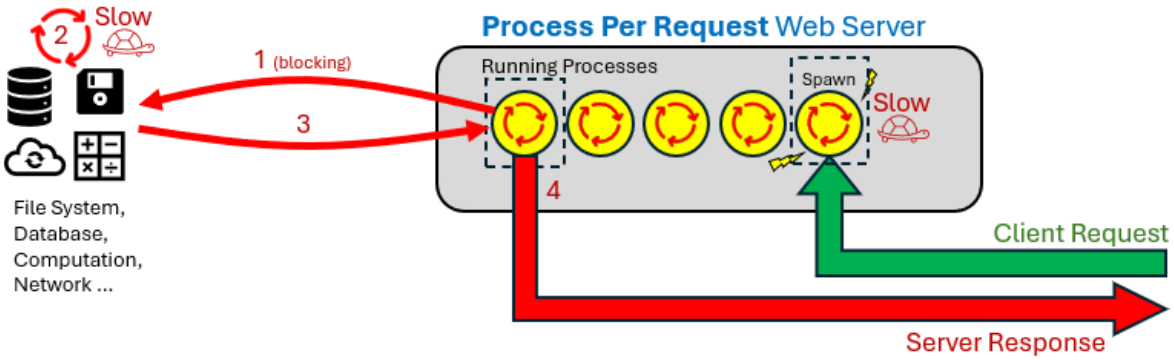


This chart visually illustrates how vastly different the response times (i.e., latency) are between various computing tasks. It shows ultra-fast cache access, much slower disk reads and even slower network communication. The point is, there is a huge bottleneck in processing time when it comes to some operations dealing with disk access and network communications. So, when a server must wait for a file to be read (or for a packet to be sent and received) before handling any other client requests, the delay can feel like “an eternity”.

Let’s compare three common server architectures:

- Process-per-request
- Thread pool-based
- Event-driven (non-blocking I/O)

(1) Originally, web servers handled each client request by spawning a separate process to handle each request as shown in the diagram below:



There are two features of this design that can cause significant delay…

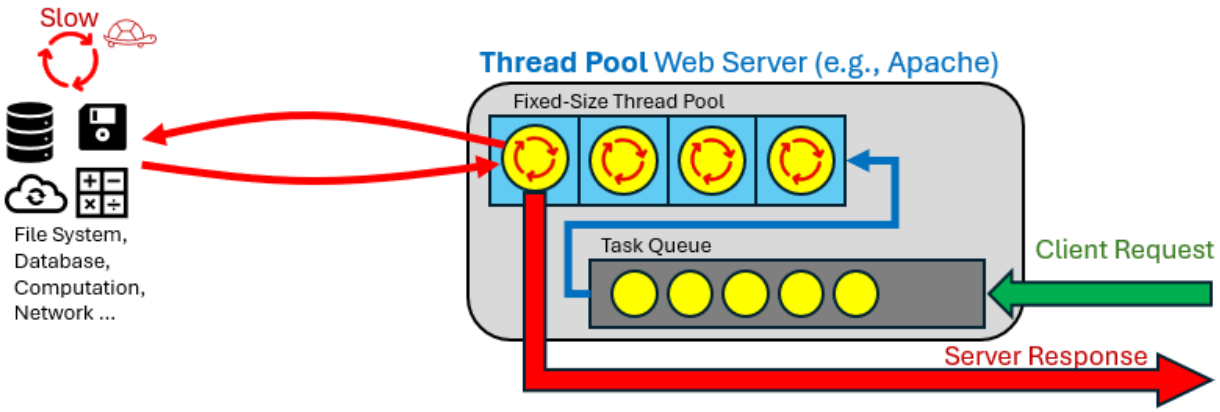
1. **Spawning** a process is considered slow because it requires the operating system to allocate separate memory, initialize resources, and set up a full execution environment.

2. Since all processes share the CPU, if one takes too long to finish, it can **hog CPU time** and delay the others. It can be compared to the notion of one slow person holding up a line. In the diagram above, it could take a while for the first process to complete due to the delay of accessing the file system or database, or due to computation-heavy calculations, or even due to slow network communications.

As an analogy, imagine a restaurant where every new customer (i.e., client request) that walks in gets their own personal waiter, cook, and table … a full dedicated staff just for them (i.e., their own process). When a customer orders food, that entire team serves only them until they're done eating and leave the restaurant. This means each client request (i.e., customer) gets its own process (i.e., staff) and they are totally isolated and independent from others.

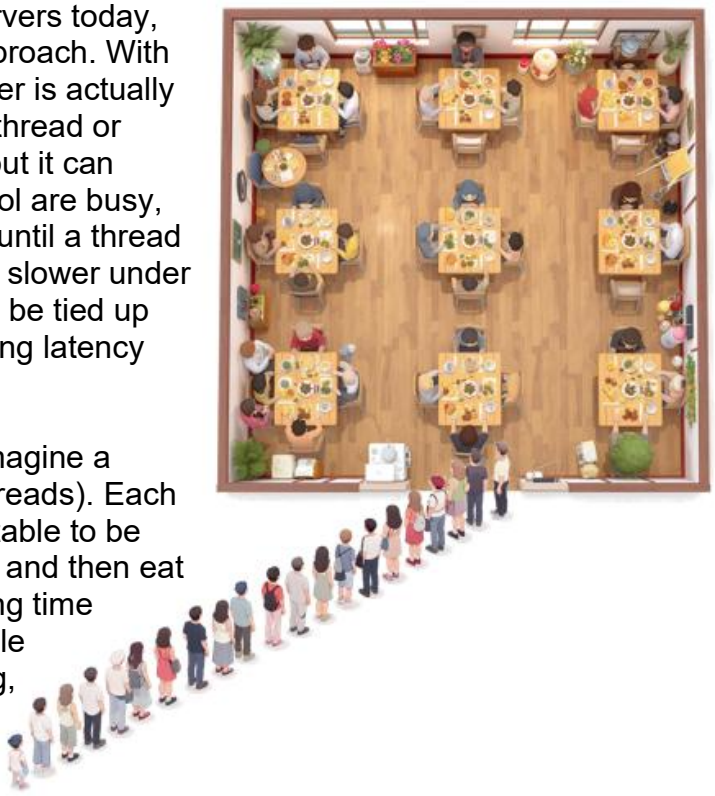
While this approach is simple and avoids interference between client requests … it is also very expensive and inefficient. For example , if 500 customers arrive, we would need 500 waiters and 500 cooks, which is not scalable.

(2) A more traditional approach is to use **threads** instead of processes. Threads are faster than processes because they share memory and resources, are cheaper to create, and switching between them can be more efficient. A thread-pool server pre-allocates a fixed number of threads to handle client requests. Incoming requests are placed in a queue and dispatched to available threads as they become free, enabling efficient reuse of resources without the overhead of creating a new thread for each request. Here is a diagram depicting such a server:



How does this compare to a **process-per-request** server? Well to use our simple restaurant analogy … a **process-per-request** server is like hiring a brand-new team of employees for each customer that comes in. A **thread pool** is like having a few small trained teams already in place, where each team quickly handles the next customer in line.

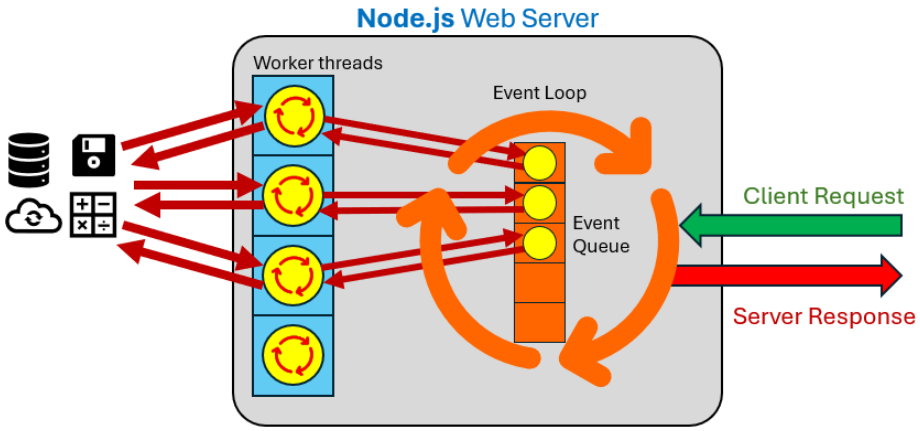
Apache is one of the most widely used web servers today, and is often configured to use a thread-pool approach. With **Apache**, all active requests (i.e., when the server is actually processing something) always get a dedicated thread or process. This approach is not inherently slow, but it can become a bottleneck when all threads in the pool are busy, and incoming requests have to wait in a queue until a thread becomes free. Thus, a thread pool can become slower under high load, because threads are limited, and can be tied up waiting hence reducing throughput and increasing latency for queued requests.



To continue our previous restaurant analogy, imagine a restaurant with a fixed number of tables (i.e., threads). Each customer (e.g., client request) is assigned to a table to be served. Once seated, the customer might order and then eat quickly (e.g., a fast task), or they may take a long time waiting for their order (e.g., waiting for I/O). While they sit there, perhaps doing nothing but waiting, the table is occupied. If all tables are full, any new customers have to wait in line until a table becomes free. This can be very inefficient especially if many tables have customers waiting for food (e.g., thread idle while waiting for I/O).

(3) A better alternative to assigning each client request its own thread is to use a single-threaded ***event loop*** that handles requests asynchronously. This is the model used by **Node.js**. When a request arrives, it's registered as a callback and the ***event loop*** continues running without blocking.

Non-blocking operations (e.g., handling **HTTP** headers, routing logic, network I/O, etc..) are processed immediately by the main thread. Blocking or time-consuming tasks (e.g., file system access, **DNS** lookups, database queries, etc..) are offloaded to a background thread pool managed by **Node.js**. If all threads are busy, additional tasks are queued in an ***event queue*** until a thread becomes available. This architecture allows **Node.js** to efficiently handle many concurrent requests with minimal overhead, using a single thread for coordination and a limited number of worker threads for heavier operations:



Going back to our restaurant analogy, **Node.js** is like a restaurant with only one Customer Service Representative (i.e., event loop) but no assigned tables for each customer. The CSR takes a customer's order, tells them to wait nearby until called (i.e.,. callback function). In the meantime, the CSR continues taking other orders without delay (i.e., iterates through event queue). When an order becomes ready, the CSR delivers the food (i.e., server response) to the customer and goes back to taking orders. The big advantage is that dozens (or hundreds) of customers can be "in progress" at any moment in time.

So where are the real savings? With time-intensive tasks like disk reads or database queries, someone, somewhere, still has to wait … the time itself doesn't go away. **Node.js** saves time not by eliminating the wait, but by ensuring that waiting doesn't block other work. In other words, **Node.js** doesn't make slow tasks faster, it just avoids letting them hold up everything else. That's where its speed and scalability come from. **Node.js** can support tens of thousands of concurrent requests in the event loop, especially for I/O-bound operations!! This is one of its biggest strengths.

One drawback of **Node.js** is that it's single-threaded at its core, so a single CPU-intensive task (such as heavy computation or blocking code) can block the event loop, preventing other requests from being processed. In extreme cases, this can cause performance degradation or even crash the application if not properly handled. This is why **Node.js** is best suited for I/O-bound workloads. For CPU-intensive tasks, it's recommended to offload the work using worker threads, child processes, or external services, to prevent blocking the event loop.

8.2 Event Loop Processing

The ***event loop*** is at the heart of how **Node.js** handles asynchronous operations while maintaining non-blocking behavior on a single thread. The loop itself is iterative and handles all callbacks.

Before the event loop starts, **Node.js** executes **ALL** our synchronous code. This includes variable declarations, function calls, `console.log()` statements, and so on. If the code includes asynchronous functions with callbacks (such as `setTimeout`, `fs.readFile`, or `promise.then`), those callbacks are registered as the code is encountered … but the callbacks themselves are not executed at that time.

Once it has completed the initial execution phase, **Node.js** enters the ***event loop*** … which is an ongoing cycle where it processes asynchronous operations by executing their associated callbacks as they become ready. When the event loop queue is empty and there are no more pending operations or callbacks to process, **Node.js** will exit.

Asynchronous tasks are categorized into two main types: ***microtasks*** and ***macrotasks***. These represent different kinds of queued operations that are handled by the event loop.

- ***Microtasks*** include operations like `promise.then()`, `promise.catch()`, `await` and `queueMicrotask()` and are executed immediately **after** current synchronous code finishes.

- ***Macrotasks*** include things like `setTimeout()`, `setInterval()` and I/O callbacks. They are processed in specific phases of the event loop and run **after** all microtasks have been cleared.

Here is the pseudocode for **Node.js** … the **REPEAT/UNTIL** is the ***event loop:***

```
Execute all synchronous (top-level) code

REPEAT {
   Process all pending microtasks
   Process ready timer callbacks (e.g., setTimeout)
   Process pending I/O callbacks (e.g., OS-level events)
   Process new I/O callbacks (e.g., file read done)
   Process scheduled setImmediate() callbacks
   Process "close" event callbacks
} UNTIL (there is no more work to do)
```

Let's look at some examples. Can you guess the output of this **asyncTest.js** code, based on the above pseudocode?

```
function middle() {
    console.log("Middle");
}

console.log("Start");

setTimeout(function cb(){
    console.log("This will be delayed by approximately 2 seconds");
}, 2000);

middle();

setTimeout(middle, 1000);

setTimeout(function cb(){
    console.log("This will be delayed by 0 seconds");
}, 0);

queueMicrotask(() => {
    console.log("This was a Microtask");
});

console.log("End");
```

```
Start
Middle
End
This was a Microtask
This will be delayed by 0 seconds
Middle
This will be delayed by approximately 2 seconds
```

Notice that all synchronous code is evaluated first. Then the microtask that was set up, then the timeout callbacks, based on their timer values.

Now, consider the following code from **badCode.js**, which has a CPU-intensive operation:

```
console.log("Start");

function fibonacci(n) {
    if (n < 2)
        return 1;
    else
        return fibonacci(n - 2) + fibonacci(n - 1);
}

console.time("timer");        // start the timer

setTimeout(function () {
    console.timeEnd("timer"); // stop the timer and display
}, 1000);

fibonacci(44);                // CPU-intense operation

console.log("End");
```

```
Start
End
timer: 5.622s
```

There is only one timer callback set up. The `console.time()` sets up a timer labelled `"timer"`. The `console.timeEnd()` ends the specified timer and displays the amount of time that the timer was running for. Look at the output. There is actually a long delay after `Start` is displayed because the `fibonacci()` function is CPU-intensive and is synchronous (i.e., blocking).

So, nothing was happening while the function was being evaluated. Therefore, even though we set up our code to end the timer after **1** second, it still took over **5.6** seconds before that timer code was evaluated, due to the blocking nature of the synchronous call to the `fibonacci()` function.

This example highlights two important warnings:

1. Timers are not guaranteed to run our code at the time we specify.
2. CPU-Intensive synchronous code is a bad idea since it blocks everything (including the event loop) from processing anything!

Now let's consider reading a file or simulating a slower web request in our **fileAndWeb.js** file. Look at this code and see if you can guess at the output (assuming the **index.html** file is our simplified FutureTechCorp. homepage) ?

```
const fs = require("fs");

console.log("Start");

fs.readFile("index.html", 'utf8', function(err, data){
    if(err) {
        console.log("Error :(");
        console.log(err);
    } else {
        console.log("File Read Successfully!");
        console.log(data);
    }
});
```

```
console.log("Middle");

// simulate a delayed operation based on the specified number of seconds
function longRunningOperation(callback, delay) {
    setTimeout(callback, delay);
}

function webRequest(request) {
    console.log('starting a long operation for request:', request.id);
    longRunningOperation(function () {
        console.log('ending a long operation for request:', request.id);
    }, request.delay);
}

// simulate a web request
webRequest({ id: 1, delay: 3000});

// simulate a second web request
webRequest({ id: 2, delay: 2000});

console.log("End");
```

The output is as follows … was it what you expected?

```
Start
Middle
starting a long operation for request: 1
starting a long operation for request: 2
End
File Read Successfully!
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>FutureTech - The Future is Now</title>
  <link rel="icon" href="icons/logo-icon.png">
  <link rel="stylesheet" href="styles/image-slider.css">
  <link rel="stylesheet" href="styles/general-body.css">
  <link rel="stylesheet" href="styles/header-footer.css">
</head>

<body>
  <header>
    <img src="images/logo.jpg" height=200 alt="FutureTech Corp. Logo">
    <nav>
      <ul>
        <li><a href="index.html" class="active">Home</a></li>
        <li><a href="about.html">About Us</a></li>
        <li><a href="products.html">Products</a></li>
        <li><a href="contact.html">Contact Us</a></li>
      </ul>
    </nav>
  </header>

  <br>

  <div style="text-align: center;">
    <img src="images/news/android.jpg" alt="Android Development Lab">
    <div class="caption">Android Development Lab</div>
  </div>
```

```
    <footer>
      <p>&copy; 2025 FutureTech. All rights reserved.</p>
    </footer>

  </body>
  </html>
  ending a long operation for request: 2
  ending a long operation for request: 1
```

Now, we are starting to get a feel for how the **Node.js** event loop processes tasks in order. It's also helpful to understand how the ***event queue*** interacts with the ***thread pool***, which is responsible for handling I/O-intensive operations. By default, **Node.js** is configured with **4** worker threads in the pool. When an I/O-heavy task (e.g., a file read or a crypto operation) is triggered, it's offloaded to one of these threads. The first four such tasks are assigned immediately. But then since all threads are already busy, any additional tasks are placed into the event queue and must wait for a thread to become available before they can begin.

Let's look at an example that makes this clear. We will set up **10** calls to a built-in **Node.js** function used for password-based key derivation which is called `crypto.pbkdf2()`. It belongs to the `"crypto"` module, which provides cryptographic functionality. Do not concern yourself with what this function does … just know that it is a *time-intensive* operation. Here is the code:

```
const crypto = require("crypto");

console.log("Start");

for (let i=0; i<10; i++) {
    console.time(`Task ${i}`);
    crypto.pbkdf2("password", "salt", 110_000, 64, "sha512", () => {
        console.timeEnd(`Task ${i}`);
    });
}

console.log("All crypto tasks started");

console.log("End");
```

The code simply calls the CPU-intensive function **10** times and records the end time of each process. Here is the output on the right →

```
Start
All crypto tasks started
End
Task 1: 81.407ms
Task 3: 83.316ms
Task 0: 87.671ms
Task 2: 88.302ms
Task 4: 163.688ms
Task 7: 165.821ms
Task 5: 170.93ms
Task 6: 173.963ms
Task 8: 233.134ms
Task 9: 234.555ms
```

Notice that the first 4 calls complete roughly at the same time (although not in the same order that we started them). The next 4 tasks had to be queued, so they took roughly twice the time to complete because they started later. The last two tasks had to wait even longer because they had to wait for two rounds of the threads to complete before becoming available.

The example is simple, but it helps us visualize the effects of queuing events/requests.

8.3 The **Module System** of Node.js

Because we have programmed in Java, we know how to organize code into classes and packages. In C, we organize reusable code into functions in header (**.h**) and source (**.c**) files, then include them where needed. **Node.js** ***modules*** work similarly. They let us break our code into reusable files, where each file is like a self-contained unit with functions, variables, or objects that we can use elsewhere. Like classes, ***modules*** help us to stay organized and make our code more readable. It also promotes code-reuse.

There are three types of ***modules*** in **Node.js**:

1. **Built-in modules** - similar to `java.util.*` in JAVA or `<stdlib.h>` in C

   e.g., **`fs`**, **`http`**, **`path`**, **`crypto`**

2. **File modules** - similar to our own **`.java`** or **`.c`** source code files

   e.g., our own **`.js`** files

3. **External modules** - similar to importing external **`.jar`** libraries or linking C libraries

   e.g., **`express`**, **`lodash`**, etc..

We use the **`require()`** function to inform **Node.js** that we require a particular module for our code. It is like asking **Node.js**: "Go find this file/ module and give me whatever it exports so I can use it here". It is similar to the idea of doing an **`import`** `java.util.ArrayList;` in JAVA or doing `#include <stdlib.h>` and linking its **lib** in C.

Often we store the module in a variable, with a similar name:

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

We have been using some modules already.

Node.js automatically treats EVERY file as a module. Behind the scenes, it creates a **`module`** object for each file. This object includes a special property called **`exports`**, which is used to define what parts of the file (e.g., variables, constants, or functions) should be exported and made accessible to other files via **`require()`**. An exported function in **Node.js** is kinds like a static method in Java iin t hat we don't need to make an object to use it. But in Java, the container is always a class, while in **Node.js** the container is the file (module).

Here is an example of a file called **employeeUtils.js** that contains some constants and functions that are all exported (i.e., made available to be used by others).

```
// Constants
const COMPANY_NAME = "TechNova Inc.";
const MAX_EMPLOYEES = 100;

// Check if an employee is full-time
function isFullTime(hoursWorked) {
    return hoursWorked >= 35;
}
```

```
// Return a string summary of an employee
function getEmployeeSummary(name, hoursWorked) {
    const status = isFullTime(hoursWorked) ? "Full-Time" : "Part-Time";
    return `${name} works at ${COMPANY_NAME} as a ${status} employee.`;
}

// Return a single employee object
function createEmployee(id, name, hoursWorked) {
    return {
        id,
        name,
        hoursWorked,
        status: isFullTime(hoursWorked) ? "Full-Time" : "Part-Time"
    };
}

// Return a list of employees
function getAllEmployees() {
    return [
        createEmployee(1, "Sanju", 40),
        createEmployee(2, "Jin", 28),
        createEmployee(3, "Alfredo", 36)
    ];
}

// Export everything (order doesn't matter)
module.exports = {
    COMPANY_NAME,
    MAX_EMPLOYEES,
    isFullTime,
    getEmployeeSummary,
    createEmployee,
    getAllEmployees
};
```

How do we make use of this module from some other code? We just use the `require()` function call and then access the components (i.e., variables, constants or functions) by using the dot operator.

Here is a **moduleUsage.js** that makes use of the module that we just exported above. Note that we are specifying that the module is in the same directory/folder as this **moduleUsage.js** file with the `"./"` at the beginning of the module name.

```
const emp = require("./employeeUtils");

console.log(emp.COMPANY_NAME);
console.log(`Max employees: ${emp.MAX_EMPLOYEES}`);

let priyal = emp.createEmployee(101, "Priyal", 42);
console.log(priyal);

let allEmps = emp.getAllEmployees();
console.log(allEmps);

console.log(emp.getEmployeeSummary("Jin", 28));
```

Here is the output for this code:

```
TechNova Inc.
Max employees: 100
{ id: 101, name: 'Priyal', hoursWorked: 42, status: 'Full-Time' }
[
  { id: 1, name: 'Sanju', hoursWorked: 40, status: 'Full-Time' },
  { id: 2, name: 'Jin', hoursWorked: 28, status: 'Part-Time' },
  { id: 3, name: 'Alfredo', hoursWorked: 36, status: 'Full-Time' }
]
Jin works at TechNova Inc. as a Part-Time employee.
```

The `require()` function in **Node.js** is synchronous, meaning it blocks execution until the requested module is loaded. So, under certain circumstances, `require()` can be slow, but this typically only applies to the first time a module is loaded. To avoid repeated file system access and improve performance, **Node.js** caches modules after the first time they're required. This means that if multiple files use `require()` to load the same module, they all receive a reference to the same single instance of that module, not separate copies.

This has important consequences, especially when `module.exports` is an object. Since objects are passed by reference in **JavaScript**, any changes made to the exported object from one part of the program will be visible everywhere else that imported it.

As an example, assume that we define a value like this in a module called **shared.js**:

```
module.exports = { counter: 0 };
```

Then we use it like this in some **JavaScript** file called **test1.js**:

```
const shared = require("./shared");
shared.counter += 1;
```

And then we use it again in another **JavaScript** file (running in the same **Node.js** process) called **test2.js**:

```
const shared = require("./shared");
console.log(shared.counter);
```

The counter will print 1, not 0.

So, be careful … even if you heard the expression:

"sharing is caring"

this can be a source of headaches if you are unaware of it.

As we have already seen previously, to use built-in modules, we also use the `require()` function but we do not include the relative path (i.e., leave out the `'./'` at the beginning of the module name).

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

Here is a table that shows some of the common built-in modules, along with a brief description:

Module	Description
`fs`	Provides functions for working with the **file system** (reading/writing files)
`path`	Helps us work with **file and directory paths** in a cross-platform way
`os`	Gives information about the **operating system** (e.g., memory and CPU)
`http`	Lets us create **HTTP servers** and handle web requests and responses
`https`	Similar to `http`, but for handling **secure HTTPS connections**
`url`	Parses and formats **URL strings** easily
`events`	Allows us to **create and handle custom events** using the EventEmitter class
`stream`	Provides tools to **work with streaming data** (e.g., file or network streams)
`util`	Offers **helper functions** (e.g., type checking, formatting, etc..)
`crypto`	Provides cryptographic features (e.g., **hashing, encryption, key generation**)
`child_process`	Lets us **run external commands or scripts** from within our **Node.js** app
`timers`	Used internally for setTimeout, setInterval, etc., but accessible if needed
`dns`	Provides functions to **perform DNS lookups** and **name resolution**
`net`	Enables creation of **lower-level TCP servers and clients**
`readline`	Lets us create **interactive command-line interfaces** (e.g., prompt user input)
`zlib`	Provides functions to **compress and decompress data** (e.g., gzip, deflate, etc.)
`assert`	Offers simple **assertion functions** for writing tests and validating conditions

There are also many external modules which are organized through a **Node Package Manager** (**NPM**). We will talk about this soon. Here are just a few of the most commonly-used:

Module	Description
`express`	Web framework for building **API**s and server-side apps quickly and easily
`dotenv`	Loads environment variables from a .env file into process.env
`axios`	Promise-based **HTTP** client for making **API** requests (both in **Node.js** and the browser)
`nodemon`	Dev tool that auto-restarts our app when we make changes to the source code
`jsonwebtoken`	Implements **JSON** Web Tokens (JWT) for secure user authentication and authorization
`mongoose`	**ODM** library for managing **MongoDB** data using schemas and models
`cors`	Middleware to enable Cross-Origin Resource Sharing in **Express** apps

8.4 Creating Servers

Now that we understand communications between clients and servers as well as the use of modules, let's see how to create a very simple server and get it running. We can make use of the `http` module. This module has a `createServer()` function that takes a callback function as a parameter. The callback function should have two parameters … the first represents the incoming `request` object from the client and the second represents the `response` object to be returned to the client.

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

let server = http.createServer((request, response) => {
  // Write code here to handle the incoming requests and generate responses
});
```

Once this is set up, we just tell the server to start listening to incoming client requests by calling the `listen()` function, passing in a port number. We will use **3000** as a port number regularly in this course. It is the call to `listen()` that actually starts the server:

```
const PORT = 3000;
server.listen(PORT);
console.log(`Server running at http://localhost:${PORT}`);
```

Assuming that we save this code in a file called **serverTemplate.js**, to run it we can just go to the VScode terminal window and type: **`node serverTemplate.js`**. This will start the server. We should see the single line appear and our code will be in an infinite loop (which we can stop with Ctrl-C):

```
Server running at http://localhost:3000/);
```

Once the server is running, we can open a browser and go to this address to access our server:

`http://localhost:3000/`

However, at this time, our basic server template does not do anything with the incoming requests, so it will not respond.

Here is another way to write the code that is cleaner, since we do not require the server variable:

```
let http = require("http");
let PORT = 3000;

function requestListener(request, response) {
      // Write code here to handle the incoming requests and generate responses
}

http.createServer(requestListener).listen(PORT);
console.log(`Server running at http://localhost:${PORT}`);
```

Now, let's add some code to make things more interesting.

Heads Up: as a side point for future reference … if we do any updates to the server while it is running, we will not see our updates applied until we restart the server and refresh the localhost page. So, stop the server with Ctrl+C, start the updated server and then refresh the localhost page in our browser.

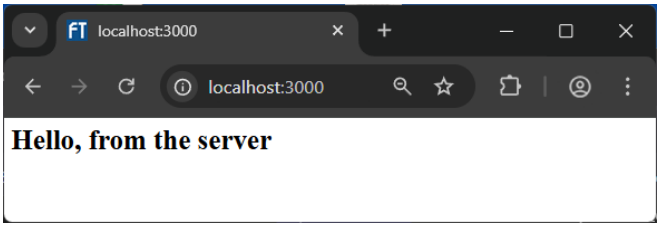
To see anything happen in the browser, we need to respond to the request by sending something back. Let's send back the standard **200 OK** response along with a simple html text message. To do this, we need to write to the `response` object and read a couple of things from the `request` object. We can use `response.writeHead()` to write the response header and the use `response.write()` to write to the body. Finally, we use `response.end()` to send the response.

Here is what we will do in our **basicServer.js** file:

```
let http = require('http');
let PORT = 3000;

function requestListener(request, response) {
    console.log("Server: I received a request for ", request.url);
    response.writeHead(200, {"Content-Type": "text/html"}); // OK status and html text response
    response.write("<h1>Hello, from the server</h1>");      // html text body to send back
    response.end();                                         // send the response now
}

http.createServer(requestListener).listen(PORT);
console.log(`Server running at http://localhost:${PORT}`);
```

Notice that we are first displaying the `request.url` information. This is the path/file that is being requested from the client. We then write the head by supplying the response status number and the headers object that contains the header information that we want to return. In this case, we at least want to tell the client that the body content is of type `"text/html"`. The body itself is just a simple html text message.

We can start this new server and then access it from our browser. This is what the browser will show:



Yay! It worked! If we go back to the VScode terminal window and look at the console output, we will notice that the browser actually sent two requests … one for the main page and one for the favicon:

```
Server running at http://localhost:3000/
Server: I received a request for  /
Server: I received a request for  /favicon.ico
```

So, a reload from the Chrome browser generates two requests by default.

That was not too bad. Now let's see how to get our **FutureTech Corp.** website up and running as a server on the local host. To do this, we need our server to be able to send back the necessary **HTML**, **CSS**, **JavaScript**, image files and logos whenever the browser asks for them. So, we will need to add a little more to our basic server above. We will need to be able to access the file system so that we can send the files to the client browser when they are requested. We can add these modules to get access to **Node.js**'s file and path-related functions:

```
let fs = require("fs");
let path = require("path");
```

Each time we send a file, we will need to set the **Content-Type** appropriately. So, we could create a simple array that maps the **Content-Type** to the file extension, for each type of file that we have in the server files that we created. These content types are also known as ***MIME*** types(***Multipurpose Internet Mail Extensions***). Originally designed for email, MIME defines a way to specify the type and format of content being sent over the internet. We can create the following object for our use, which contains a file extension mapping to content types we will need for the header:

```
const mimeTypes = {
    ".html": "text/html",
    ".css":  "text/css",
    ".js":   "application/javascript",
    ".png":  "image/png",
    ".jpg":  "image/jpeg",
    ".ico":  "image/x-icon"
};
```

Here is what we have so far for our **futuretechServer.js** file:

```
const http = require("http");
const fs = require("fs");
const path = require("path");

const PORT = 3000;
const mimeTypes = {
    ".html": "text/html",
    ".css":  "text/css",
    ".js":   "application/javascript",
    ".png":  "image/png",
    ".jpg":  "image/jpeg",
    ".ico":  "image/x-icon"
};

function requestListener(req, res) {
    // the FUN stuff goes here
}

http.createServer(requestListener).listen(PORT);
console.log(`Server running at http://localhost:${PORT}`);
```

Now, all that needs to be done is to complete the listener to handle the requests. When the browser requests a file, we send it back, provided that there was no error reading it. Remember … when a browser requests a site, it first wants the **HTML** file sent back. Then it will start requesting the images, icons, **CSS** and **JavaScript** files one at a time. So, we need to be able to handle all of these.

Also, it is rare that the user would request specifically the default **index.html** file, which is the main page. So, if they simply go to our site, with no specific file specified (i.e., the requested **URL** is `"/"`), then we need to send back the **index.html** file. Even then, once they specify the file that they want, we still need to get that file in our server's file system with the absolute path to that file. So, here is how we will begin:

```
function requestListener(req, res) {
  let filePath = req.url === "/" ? "/index.html" : req.url; // add index.html if needed
  filePath = path.join(__dirname, filePath);                // get absolute path to file

}
```

This code looks at the request's **URL** and changes it to `"/index.html"` if needed. Then it uses the `__dirname` variable, which is a special variable in **Node.js** that gives us the absolute path to the directory where the currently executing script file resides. So, `filePath` is set to the absolute path of the requested file. Now we know where to find the file 😊.

Next, let's look at the file extension and set up a mime type that we can send back in the header. The `path` module's `extName()` function lets us extract the extension for the file requested (e.g., `".jpg"`). We can then find the content type for that extension by looking it up in our `mimeTypes` object mapping. If, for some reason, we do not find the file type that matches, we can set the content type to the general: `"application/octet-stream"`. So, we add this code to our listener:

```
  const ext = path.extname(filePath);  // get the file extension

  // lookup content type based on ext. If not there, treat as a download
  const contentType = mimeTypes[ext] || "application/octet-stream";
```

By using the OR (i.e., `||`) operator, we are essentially saying: "If the file type was not found in the object (i.e., returns **`undefined`**), then use this one instead."

OK. Now we are ready to read in the requested file and send it back to the client. The read will either be successful or it will fail (e.g., file not found). So, we must handle both of those situations.

The `fs` module contains the function that we need to read the file. The module's `readFile()` function takes the filename as its first parameter and a callback function as its second parameter.

Let's see if you have been paying attention. Why is a callback function required? Well, as we well know by now, this is an I/O operation which can take time to perform. Therefore, it will not be processed right away but will be assigned to a thread in the pool. When the file read comes back successfully, the event loop will be notified and it will then call our callback function to send the information back to the client.

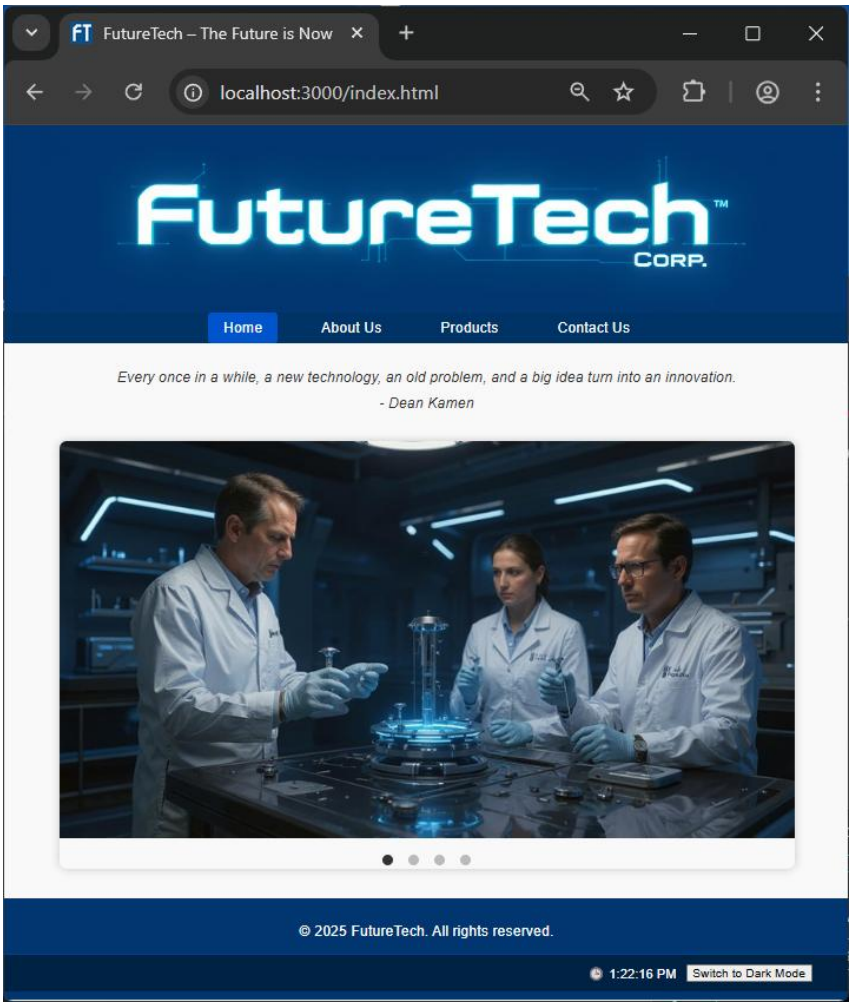
The callback function that we supply must take two parameters. The first is an object that will be **null** if all went well, or an object if an error occurred. The second parameter is the object that represents data from the file. In our case, it will be the binary data of the file.

So, here is what we need to do …

```
fs.readFile(filePath, (err, data) => {                  // read file, set data to contents
  if (err) {                                            // if error, return 404 Not Found
    res.writeHead(404, { "Content-Type": "text/plain" });
    return res.end("404 Not Found");
  }
  res.writeHead(200, { "Content-Type": contentType });    // otherwise send 200 OK and the data
  res.end(data);
});
```

From the above code, we can see that if an error occurs, we send back a **404** response in the header with a content type saying that it is plain text. Then we include a simple string in the body indicating the error. Notice that we are supplying a parameter to the `res.end()` function. This will write that parameter as the body of the response and then send the response. In the case of no error, we send back the **200** response and use our `contentType` variable (that we set earlier) to indicate the kind of body content. We then send back the `data` from the file reading.

That's it! Now we can run this with **Node.js**. The result is that we can go to a browser on our laptop/desktop and type in **localhost:3000** … and then it will open up our webpage and we can navigate the page.



8.5 Handling Request Data

Lat's talk more about the request and response parameters for our listener callback function. The request object has some useful properties:

- `request.method` – the **HTTP** method of the request (i.e., **`GET`** or **`POST`**)
- `request.url` – the **URL** of the request (e.g., **`/index.html`**)
- `request.headers` – an object containing all headers

As an example, consider this **requestDetails.js** server code:

```
const http = require("http");
const PORT = 3000;

function requestListener(request, response) {
    console.log("method: " + request.method);
    console.log("URL: " + request.url);
    console.log("headers: " + JSON.stringify(request.headers));
    response.end();   // an empty response sent back
}

http.createServer(requestListener).listen(PORT);
console.log(`Server running at http://localhost:${PORT}`);
```

When we access the server by going to **localhost:3000** we get the following information written to the console (and we send back a response with no header nor body):

```
method: GET
URL: /
headers: {"host":"localhost:3000","connection":"keep-alive","cache-control":"max-
age=0","sec-ch-ua":"\"Not)A;Brand\";v=\"8\", \"Chromium\";v=\"138\", \"Google
Chrome\";v=\"138\"","sec-ch-ua-mobile":"?0","sec-ch-ua-platform":"\"Windows\"","upgrade-
insecure-requests":"1","user-agent":"Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/138.0.0.0
Safari/537.36","accept":"text/html,application/xhtml+xml,application/xml;q=0.9,image/avif
,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7","sec-fetch-
site":"none","sec-fetch-mode":"navigate","sec-fetch-user":"?1","sec-fetch-
dest":"document","accept-encoding":"gzip, deflate, br, zstd","accept-language":"en-
US,en;q=0.9"}
request starting...
method: GET
URL: /favicon.ico
headers: {"host":"localhost:3000","connection":"keep-alive","sec-ch-ua-
platform":"\"Windows\"","user-agent":"Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/138.0.0.0 Safari/537.36","sec-ch-
ua":"\"Not)A;Brand\";v=\"8\", \"Chromium\";v=\"138\", \"Google Chrome\";v=\"138\"","sec-
ch-ua-
mobile":"?0","accept":"image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8",
"sec-fetch-site":"same-origin","sec-fetch-mode":"no-cors","sec-fetch-
dest":"image","referer":"http://localhost:3000/","accept-encoding":"gzip, deflate, br,
zstd","accept-language":"en-US,en;q=0.9"}
```

As we can see, there is a lot of information available in the header **JSON** object that is returned. Our server can examine these headers to determine what is being requested and to make some decisions as to what it should return.

Remember as well that **GET** requests do not have a body. If there is any data coming along with the request, it will be coming within the **URL** string:

`http://localhost:3000/index.html?year=2022&month=october#content`

Do you remember how to parse this?

```
const url = require("url");

let q = url.parse(request.url, true);
let qdata = q.query;        // returns an object: { year: 2022, month: 'october' }
```

So here is a **requestQueryDetails.js** server that will display the query information supplied with any incoming request:

```
const http = require("http");
const url = require("url");

const PORT = 3000;

function requestListener(request, response) {
    console.log("URL: " + request.url);

    let q = url.parse(request.url, true);
    let qdata = q.query;        // returns an object: { year: 2022, month: 'october' }
    for (x in qdata) {
        console.log(x + ": " + qdata[x]);   // year: 2022 and month: october
    }

    response.end();  // an empty response sent back
}

http.createServer(requestListener).listen(PORT);
console.log(`Server running at http://localhost:${PORT}`);
```

If we go to this webpage:

http://localhost:3000/index.html?year=2022&month=october#content

We get this output:

```
Server running at http://localhost:3000
URL: /index.html?year=2022&month=october
year: 2022
month: october
URL: /favicon.ico
```

Now, what if our request type is a **POST** or **PUT** instead of a **GET**? In these cases, the incoming request will contain a body. Reading the body requires a bit more work because it involves additional I/O, which is handled asynchronously using callbacks. Fortunately, we can treat the request object as a readable stream and collect the data as it arrives.

A readable stream emits two important **events** we can handle:

- **data** – triggered whenever a new chunk of data is available
- **end** – triggered when all data has been received

We can attach handlers to these events to read the entire request body. The **data** event collects chunks as they arrive, so it is triggered multiple times. We just need to keep appending the latest chunk to what we have accumulated so far. Once the **end** event is triggered, that signals us that the data is all here now and we can process the full body.

The data coming from the stream is a **Buffer** object in **JavaScript**, which is essentially a raw sequence of bytes. The way that we initialize the buffer and append to it will depend on how we are using the buffer. For example, we may treat the buffer as a string of text characters, or it might be binary data (e.g., files, images, etc..). Here are two ways to initialize a `body` variable to hold the buffer data:

```
let body1 = "";                 // when the body will be a string
let body2 = Buffer.alloc(0);    // when the body will be binary data
```

Now, we just need to set up two callbacks one for when some data arrives, and one for when it is all done. Registering event handlers (i.e., callbacks) is easily done by using the `.on()` function which works for the **HttpRequest** object (as well as many other objects defined in the **Node.js** core modules). It takes, as the first parameter, the name of the event that we want to listen for (i.e., `"data"` or `"end"` in our case here) and the callback function as the second parameter. The callback function for the **data** event should have one parameter to represent the incoming data and the callback for the **end** event needs no parameters.

Here is how we would set things up to read the body of an incoming **POST** or **PUT** request if it was text data:

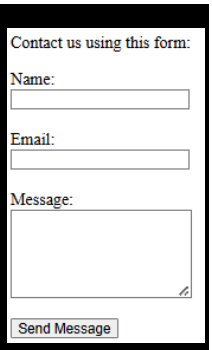
```
let body = "";                  // when the body will be a string

request.on("data", chunk => {
    body += chunk;              // chunk is a Buffer, but += calls .toString() on it
});

request.on("end", () => {
    console.log("Final body (as string):", body);
});
```

And here is how it would be set up if the body was a file of some kind (e.g., image, icon, binary file):

```
let body = Buffer.alloc(0);     // when the body will be binary data

request.on("data", chunk => {
    body = Buffer.concat([body, chunk]);
});

request.on("end", () => {
    console.log("Final body (as Buffer):", body);
});
```

Let's do an example. Here is a simple html file called **simple-form.html**, that contains a form that will be submitted to our local host server when the user presses the "Send Message" button:

```
<html>
    <head>
        <title>Sending Form Data</title>
    </head>
    <body>
        <p>Contact us using this form:</p>

        <form action="http://localhost:3000/" method="POST" target="_blank">
            <label for="name">Name:</label><br>
            <input type="text" id="name" name="name" required><br><br>

            <label for="email">Email:</label><br>
            <input type="email" id="email" name="email" required><br><br>

            <label for="message">Message:</label><br>
            <textarea id="message" name="message" rows="5" required></textarea><br><br>

            <button type="submit">Send Message</button>
        </form>
    </body>
</html>
```

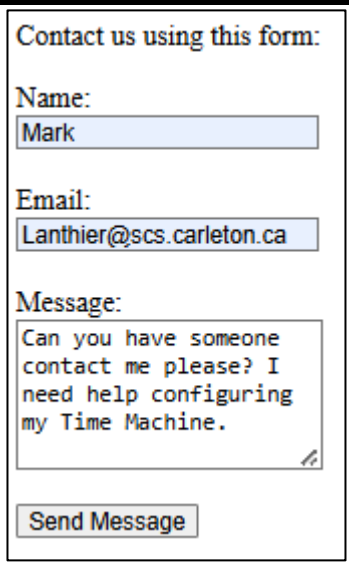


The code is set up with `target="_blank"` so that the response from the server (whatever it may be) is displayed in a new browser tab. The request itself will be sent as a default **POST** message.

Now, let's work on the server by making use of our `request.on()` code from above. Since we are using the standard form submission from our **HTML** page, we will want to make use of the `querystring` module, which lets us do `querystring.parse(body)` to access the individual input fields of the form by their names: "name", "email" and "message". Here is the code for **requestFormHandler.js**:

```
const http = require("http");
const querystring = require("querystring");

const PORT = 3000;

function requestListener(request, response) {
    console.log("method:  "  + request.method);
    console.log("URL:     "  + request.url);
```

```
    console.log("headers: " + JSON.stringify(request.headers));

    if (request.method === "POST"){
        let body = "";
        request.on("data", chunk => {
            body += chunk;
        });
        request.on("end", () => {
            console.log("Final body:", body);
            let formData = querystring.parse(body);
            console.log("Name:    " + formData.name);
            console.log("Email:   " + formData.email);
            console.log("Message: " + formData.message);

            // Send back an acknowledgement
            response.end("Thank you for your message.");
        });
    }
    else // send back nothing if it wasn't a POST
        response.end();
}

http.createServer(requestListener).listen(PORT);
console.log(`Server running at http://localhost:${PORT}`);
```

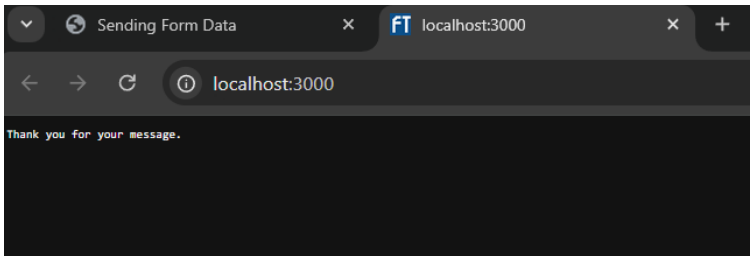


Assume that we entered information into the form (as shown on the image here to the right) and then pressed **Send Message**. The output below is what we would see in the server console:

```
Server running at http://localhost:3000
method:  POST
URL:     /
headers: {"host":"localhost:3000","connection":"keep-alive","content-
length":"133","cache-control":"max-age=0","sec-ch-ua":"\"Not)A;Brand\";v=\"8\",
\"Chromium\";v=\"138\", \"Google Chrome\";v=\"138\"","sec-ch-ua-mobile":"?0","sec-ch-ua-
platform":"\"Windows\"","upgrade-insecure-requests":"1","user-agent":"Mozilla/5.0
(Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/138.0.0.0
Safari/537.36","origin":"null","content-type":"application/x-www-form-
urlencoded","accept":"text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,im
age/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7","sec-fetch-
site":"cross-site","sec-fetch-mode":"navigate","sec-fetch-user":"?1","sec-fetch-
dest":"document","accept-encoding":"gzip, deflate, br, zstd","accept-language":"en-
US,en;q=0.9"}
Final body:
name=Mark&email=Lanthier%40scs.carleton.ca&message=Can+you+have+someone+contact+me+please
%3F+I+need+help+configuring+my+Time+Machine.
Name:    Mark
Email:   Lanthier@scs.carleton.ca
Message: Can you have someone contact me please? I need help configuring my Time Machine.
method:  GET
URL:     /favicon.ico
headers: {"host":"localhost:3000","connection":"keep-alive","sec-ch-ua-
platform":"\"Windows\"","user-agent":"Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/138.0.0.0 Safari/537.36","sec-ch-
ua":"\"Not)A;Brand\";v=\"8\", \"Chromium\";v=\"138\", \"Google Chrome\";v=\"138\"","sec-
ch-ua-
mobile":"?0","accept":"image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8",
```

```
"sec-fetch-site":"same-origin","sec-fetch-mode":"no-cors","sec-fetch-dest":"image","referer":"http://localhost:3000/","accept-encoding":"gzip, deflate, br, zstd","accept-language":"en-US,en;q=0.9"}
```


In the browser, we would see a new tab opened up with the response from our server in it as shown here →



We did not handle all types of errors in our code above. The idea was to show us how to handle a **POST**. Ultimately, our goal is to understand how to write servers to handle all possible requests. We will learn more as the course goes on, and as we practice.

8.6 Node Package Manager (NPM)

NPM stands for ***Node Package Manager***. We can find more about it here: https://www.npmjs.com/. It is used by over 17 million developers worldwide.

It's a **tool** that helps web developers **install**, **manage**, and **share code** written by others. Think of it like an **app store**, but for code libraries and tools that developers use when building websites and web applications. We do not have to install **NPM** because it comes with **Node.js**, which we have installed already.

When we are building a website or web app, we likely will not want to write everything from scratch. For example, if we need animations … there is a package for that. If we want to build a responsive layout … we can use a framework like **Bootstrap**. If we want to handle user input easily … we can install a **form** library. **NPM** helps us quickly add these features to our project.

What is **NPM** actually "managing"? ***Packages*** … which are the building blocks that **NPM** installs and organizes. Knowing how they work will help us use **NPM** effectively to build our web applications faster and more efficiently. **NPM** is the largest software registry (library) in the world … with over **2,000,000 packages available**!

A *package* is a bundle of code (which can include one or more JavaScript **files/modules) that someone wrote to solve a problem.**

A ***package*** differs from a ***module*** in that a ***module*** is just a single **JavaScript** file whereas a ***package*** is a folder of files bundled together with a `package.json` file.

The `package.json` file stores important information about our project, including:

- metadata such as the project's **name, author, version**, etc..
- a list of **dependencies** (i.e., other packages our project needs)
- **scripts** to automate tasks (e.g., running our app or tests)
- **other settings** that help tools and developers understand our project

Developers upload these packages to **NPM**'s online registry, and we can install them in our project. Some popular **NPM** packages in web development are …

- **React** – for building user interfaces
- **Express** – for backend servers
- **Lodash** – for utility functions
- **Tailwind CSS** – for styling

Let's start by reviewing how we have been using modules so far. We have mainly worked with two types: built-in modules and modules defined in local files. Let's explore how **NPM** locates the modules we want to use in our code.

We have seen that there are two main ways of using `require()` to import a module:

- For a **local** module:

```
const myLocalModule = require("./myFolder/myLocalModule");
```

- For a **Node.js** module:

```
const lodash = require("lodash");
```

The difference between the two is the specified path (i.e., `"./myFolder/"`) for local modules. When we do not supply a path, we are not telling **Node.js** where to find the module, so it uses its module resolution algorithm to search for it.

Consider doing a `require("lodash")` in a file called `myCode.js` that has the absolute file name:

`C:/Users/steve/Documents/COMP2406/MyAssignment/myCode.js`.

Here is what **Node.js** does to look for it:

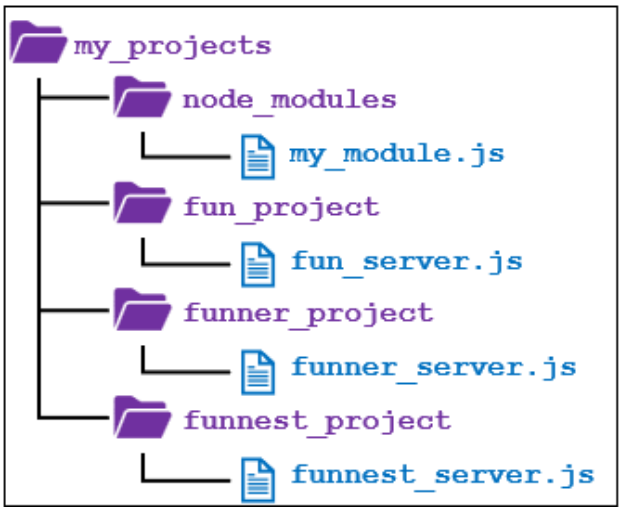
1. It first checks if `"lodash"` is a built-in module (e.g., `"fs"`, `"path"`, or `"http"`). Since `"lodash"` is not built-in, the search continues…
2. It then looks for a folder called `node_modules` in the same directory as the file that's running (i.e., it checks for `C:/Users/steve/Documents/COMP2406/MyAssignment/node_modules/lodash`). If there is no `node_modules` folder or if the `lodash` folder is not in it, the search continues…
3. It then starts looking for the module by moving up one level at a time in the directory tree. So, it would look up the tree for `"lodash"` in the following paths (in order shown):

```
C:/Users/steve/Documents/COMP2406/node_modules/lodash
C:/Users/steve/Documents/node_modules/lodash
C:/Users/steve/node_modules/lodash
C:/Users/node_modules/lodash
C:/node_modules/lodash
```

4. If it reaches the root folder without finding it, it throws an error:

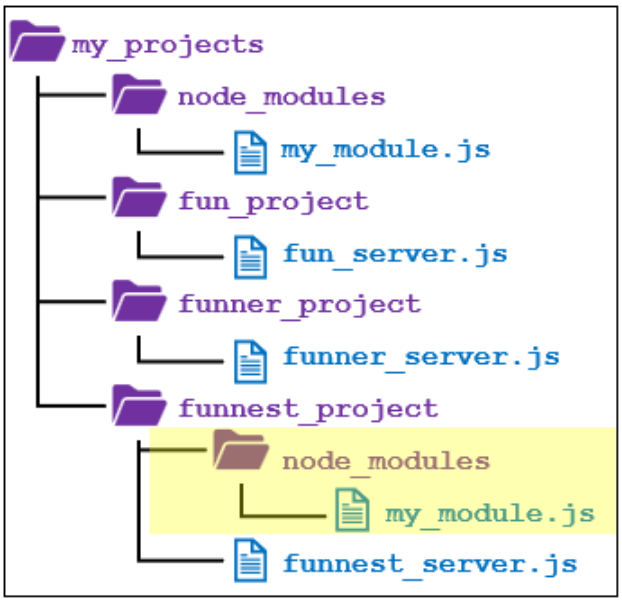
```
Error: Cannot find module "lodash"
```

It is good to understand this search order because **Node.js** programs are typically organized into projects and it can make our life simpler if we structure our folders nicely. For example, assume that we have a main **my_projects** folder with some of our own modules and then a few subfolders arranged as shown here →



If the three different project servers here all `require` the same `my_module.js`, then we can simply write `require("my_module")` in each server and they will each import the same module, due to our well-chosen directory hierarchy.

If we need another (altered) version of `my_module.js` for one specific project, we can just include a `node_modules` folder with the updated `my_module.js` file in it as shown here →



This method of relying on directory hierarchies will work, but it can (and will) get confusing for larger projects with many versions. Also, we must build the directory structure ourselves (i.e., we would have to find and download every package, "install" them, etc.). This approach is fragile and error-prone.

NPM helps us manage both our **Node.js** projects and the external code they rely on, called ***dependencies***. A ***dependency*** is simply a package (often created by someone else) that our project needs in order to work properly. These might include things like web frameworks, utility libraries, or tools for testing. All the dependencies that a project uses are listed in a file called **package.json**. This file acts like a map for our project, showing what it needs to run.

With **NPM**, we can easily install new packages, keep track of them, and re-install everything later if we move the project to a different computer. This makes our projects easier to build, maintain, and share with others.

When starting a new project, it is best practice to initialize it by running `npm init` in the project's root directory. When doing this, we will be asked some questions about the package (we can just press **Enter** until the end) and then it will create the basic `package.json` file, which will keep track of the packages that we use. We can alternatively do `npm init --yes`, which will not ask us those questions.

We **can** install packages without running `npm init` first, but if we skip it, our project won't have a `package.json` file. That file is important because it keeps track of project details and dependencies. Without it, several problems arise:

- We will have **no way to track dependencies**. If we install a package (such as `express`), it will appear in the `node_modules/` folder, but there won't be any record in the project. This means if someone else downloads our code (or if we move it to a new computer) there is no way to know which packages need to be reinstalled..

- Running `npm install` (see below) won't work, because it relies on `package.json` to know which packages to install. So, there is **no easy way to rebuild the project**.

- Other tools (e.g., **GitHub** or teammates) rely on `package.json` for metadata and dependencies. Without it, they won't know how our project is set up.

To install packages, we use `npm install`. We can install packages in two ways:

- **Local mode:** installed in a `node_modules/` directory in our current working directory.

  (e.g., `npm install moduleName`)

- **Global mode:** installed system-wide in a global `node_modules/` directory, defined by **NPM**'s `prefix` configuration.

  (e.g., `npm install moduleName -g`)

  We can find the global install location using `npm config get prefix`).

In most cases, we install packages **locally**, on a per-project basis. This means each project has its own copy of the packages it needs. While this may result in duplicate files across projects, it's a worthwhile tradeoff:

- It avoids the complexity of sharing and managing a single set of packages across many projects.
- More importantly, different projects may require different versions of the same package, and local installation makes that possible. **NPM** handles this automatically, helping each project stay consistent and isolated from others.

Whenever we install a package (e.g., `express`), that package is listed in `package.json` so that others (or even we ourselves, later) can easily recreate the environment by running `npm install`.

For example, assume that we had a `package.json` file already with the `express`, `pug` & `socket.io` modules installed already and then we do:

```
npm install underscore
```

to install the `underscore` module.

The `package.json` file will then look as shown here →

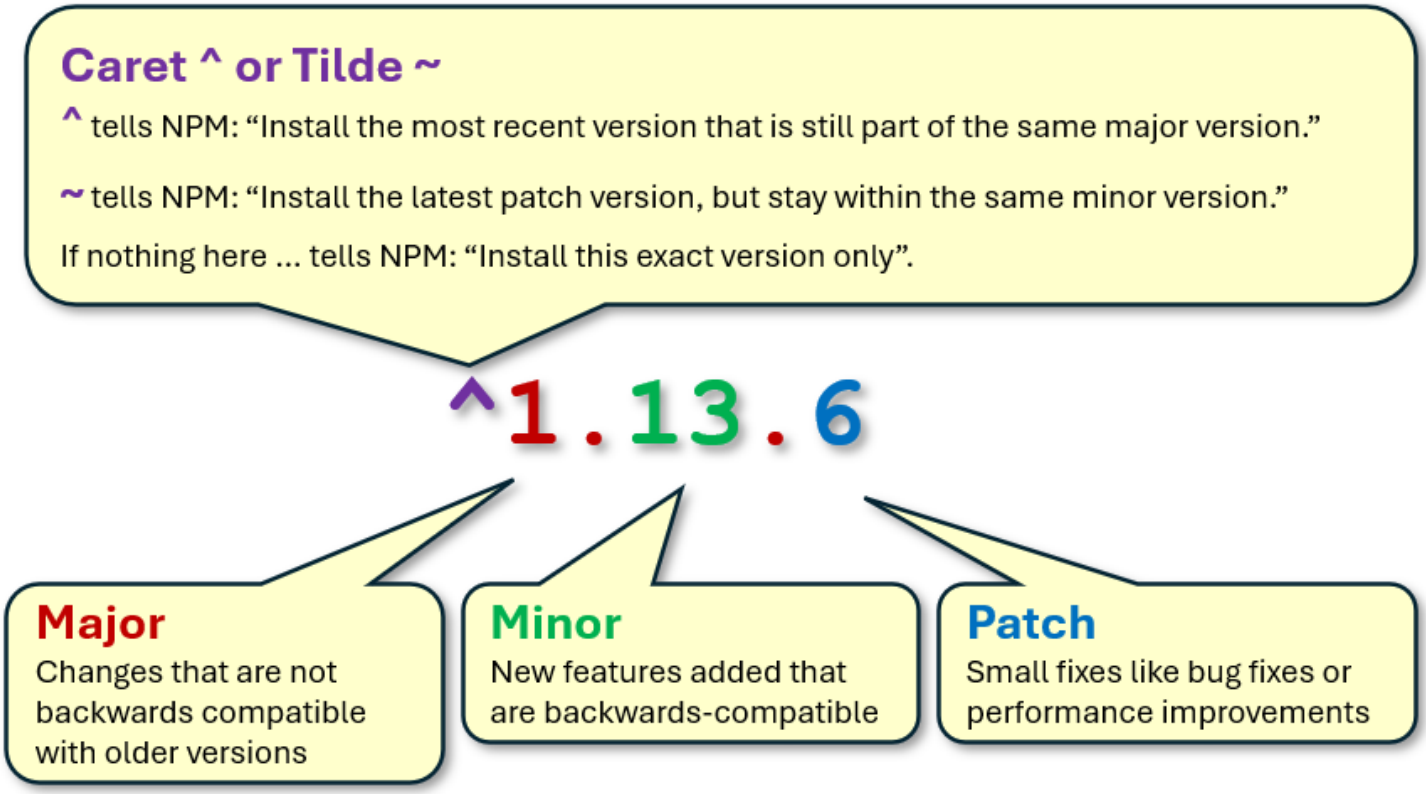
```
{
  "name": "05_01",
  "version": "1.0.0",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
  "keywords": [],
  "author": "Alina",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2",
    "pug": "^3.0.2",
    "socket.io": "^4.7.2",
    "underscore": "^1.13.6"
  },
  "devDependencies": {},
  "description": ""
}
```

When we (or a TA) are setting up a project on a new computer or environment, and need to install all required dependencies, we can simply run: `npm install` from within the directory that contains this `package.json` file. This will install all the modules listed in the file, restoring the project's full set of dependencies.

If we look at what was written to the `package.json` file, when we installed the `underscore` module, we see this:

```
"underscore": "^1.13.6"
```
← What is this?

The value of `"^1.13.6"` here is the ***semantic versioning information*** (a.k.a., ***version specifier***).



When working on a project with **NPM**, it's important that everything works the same way on every computer … whether it's ours or our TA's. A file named `package-lock.json` is automatically created by **NPM** when we install packages. It records the exact versions of every package (and sub-package) used in our project, ensuring consistent behavior across different environments. While `package.json` lists our main dependencies, `package-lock.json` locks in the specific versions actually installed. This helps avoid version mismatches and makes our project easier to share, test, and deploy.

Here are some additional things that we may want to do with **NPM**:

Install a specific version of a package

Tell **NPM** exactly which version we want to install by adding the version after the package name. For example,

```
npm install pug@3.0.0          ← Installs version 3.0.0 exactly
npm install pug@"^3.0.0"       ← Installs the latest compatible version
                                  starting from 3.0.0 (e.g., 3.0.2)
```

Remove a package

We might want to remove a package if we don't need it anymore, if it causes problems or conflicts, to keep our project smaller and cleaner, or to replace it with a better or newer package. To uninstall a package, use: `npm uninstall pug`

Update packages

We may want to do this, to get bug fixes and improvements, to add new features, to improve security, or to keep compatibility with other tools or packages. We run `npm update` to upgrade the packages installed in our project based on the version rules in our `package.json`. It updates each package to the latest version that still matches the specified range. This ensures we get new features and fixes without accidentally installing incompatible major versions.

Check installed packages

In some situations, we may want to see what packages are installed for our project: to verify package versions, to find outdated or unused packages, or to troubleshoot errors or conflicts. To check packages and versions, use: `npm list`