Backend - It refers to the server-side of an application. It's the part of a web application that deals with the underlying logic, data storage, and interaction with databases, as well as handling requests from clients (such as web browsers) and sending back responses. Backend serves as the foundation that supports the frontend, which is the user interface that users interact with directly.Java, PHP, Python, and Node.js are some of the backend development technologies

Frontend - It refers to the client side of an application. It is the part of the applicaton that users can see and interact with and includes everything that attributes to the visual aspects of a web application.html, css, javascript, reactjs etc are some examples of tech stacks which are used in the frontend.

nodejs - a framework which directly compiles the javascript without the help of browser. it is used to create server-side web applications and api for the frontend.aysnchronous and single threaded.fast, opensource, scalable and asynchronous. In short, Node.js is an open-source, cross-platform JavaScript runtime environment and library to run web applications outside the client's browser. It is used to create server-side web applications.
Node. js runs JavaScript code in a single thread, which means that your code can only do one task at a time. However, Node. js itself is multithreaded and provides hidden threads through the libuv library, which handles I/O operations like reading files from a disk or network requests.Through the use of hidden threads, Node.js provides asynchronous methods that allow your code to make I/O requests without blocking the main thread.
Handling more and more concurrent client's request is very easy.
Even though our Node JS Application receives more and more Concurrent client requests, there is no need of creating more and more threads, because of Event loop.
Node JS application uses less Threads so that it can utilize only less resources or memory

synchronous - evey statement of code get executed in a sequence one by one.

asynchronous - next statement of code will get executed when a synchrounous code blocks the execution. parallel execution.

```
setTimeout(() => {
    document.write("Let us see what happens");
}, 2000);
```

> NPM stands for Node Package Manager, responsible for managing all the packages and modules for Node.js.

why nodejs?
very fast, a javascript library so easy to understand for a web developers, can handle asynchronous

functions efficiently, have vast amount of library and frameworks which makes the development easier. event driven programming approach. Rich ecosystem. javascript everywhere.

demerits?
Since Node.js is single-threaded, CPU intensive tasks are not its strong suit.
Using callback is complex since you end up with several nested callbacks.

event driven programming - This approach uses events to trigger various functions. the events can be anything like pressing a key, clicking a mouse button. a callback function is already registered whenever an event is triggered.

event loop - handle asynchronous callbacks in nodejs.

callback funtions?
async await?
promises?

working of nodejs:Node.js accepts the request from the clients and sends the response, while working with the request node.js handles them with a single thread. To operate I/O operations or requests node.js use the concept of hidden threads.
Asynchronous
Non-blocking I/o: Non-blocking i/o means working with multiple requests without blocking the thread for a single request. I/O basically interacts with external systems such as files, databases. Node.js is not used for CPU-intensive work means for calculations, video processing because a single thread cannot handle the CPU works.

Arrow Function Example:

```
const sum = (a, b) => {
    return (a + b);
};
```

why do we use arrow function?

1. Concise Syntax
2. Lexical this Binding:
   One of the most significant benefits of arrow functions is their automatic lexical this binding. This means that an arrow function captures the value of this from its surrounding code (the enclosing function or global context), rather than having its own this context like traditional functions. This can be especially useful when dealing with callbacks and closures. Consider this example:

```
// Traditional function expression
function Counter() {
  this.count = 0;

  setInterval(function() {
    // This would refer to the global object or undefined (in strict mode)
    this.count++;
    console.log(this.count);
  }, 1000);
}

const counter = new Counter(); // Outputs NaN (Not a Number)

// Arrow function (lexical this)
function ArrowCounter() {
  this.count = 0;

  setInterval(() => {
    // 'this' here refers to the 'ArrowCounter' instance
    this.count++;
    console.log(this.count);
  }, 1000);
}

const arrowCounter = new ArrowCounter(); // Outputs 1, 2, 3, ...
```

In the traditional function expression, the value of this is not what we expect because the function inside setInterval has its own context. However, the arrow function maintains the correct this binding, making the code work as intended.

what is request and response?
Request:
The "request" in Node.js refers to the information that a client (usually a web browser) sends to the server. This information typically includes data like the URL being accessed, HTTP method (GET, POST, etc.), headers, query parameters, and sometimes a request body (for methods like POST and PUT).

In Node.js, the request object provides access to this information. This object contains properties and methods that allow you to retrieve various details about the incoming request. You can access the URL, headers, query parameters, and more from this object.

Response:
The "response" in Node.js is the data and metadata that the server sends back to the client after processing the request. This includes the HTTP status code (indicating the outcome of the request),

headers that define how the response should be interpreted, and the actual content of the response, which can be HTML, JSON, a file, etc.

In Node.js, the response object allows you to set the HTTP status code, headers, and send the response content. You can use various methods on the response object to modify the response before sending it back to the client.

what is module?
A module is essentially a JavaScript file that encapsulates related functions, classes, variables, and other code within a defined scope. Modules help in keeping your codebase organized, improving code maintainability, and enabling code reuse across different parts of your application.
Different types of modules:
Core Modules:
Node.js comes with a set of built-in core modules that provide essential functionality. These modules are available without needing to install any additional packages. Examples of core modules include fs (for file system operations), http (for creating HTTP servers and clients), path (for handling file paths), and util (for utility functions).

Local Modules:
These are the modules you create within your own application. Local modules allow you to encapsulate related code within separate files, making your application more organized and maintainable. You can export functions, classes, variables, and objects from local modules and import them into other parts of your application using require.

Third-Party Modules:
Third-party modules are packages created by the community and available on the Node.js package registry, npm (Node Package Manager). You can install and use these modules to add functionality to your application without having to build everything from scratch. Examples of third-party modules include express (for building web applications), lodash (for utility functions), and axios (for making HTTP requests).

what is expressJs and why do we use it?
Express.js is used to simplify and streamline the process of building web applications and APIs in Node.js. It provides a flexible and organized way to handle HTTP requests, manage middleware, render dynamic templates, and more.

Simplicity: Express provides a simple and intuitive API for handling HTTP requests and responses. Its minimalist design makes it easy to get started with building web applications without unnecessary complexity.

Routing: Express allows you to define routes that match specific URL patterns and HTTP methods (GET, POST, etc.). This makes it easy to organize your application's logic and handle different types of requests separately.

Middleware: Middleware functions in Express are used to perform tasks such as authentication, logging, data parsing, and error handling. Express middleware allows you to modularize your application's logic and apply it to specific routes or globally.

If we want to render a static html file then we can directly use inbuilt function of express and pass the exact path location of the file.

If we want to render a dynamic html file then we have to use template like ejs to render the the file.

what is middleware?
Middleware is a concept used in web development, including in frameworks like Express.js, to handle tasks that occur between receiving an HTTP request and sending an HTTP response. Middleware functions are functions that have access to the req (request) and res (response) objects, and they can perform various tasks such as modifying request/response data, executing code, or terminating the request-response cycle.

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

// Simulated user data (for demonstration purposes)
const users = [
  { id: 1, username: 'user1', loggedIn: true },
  { id: 2, username: 'user2', loggedIn: false }
];

// Custom authentication middleware
const authenticate = (req, res, next) => {
  const userId = req.params.userId; // Assume userId is part of the URL
  const user = users.find(user => user.id === parseInt(userId));

  if (!user || !user.loggedIn) {
    return res.status(401).send('Unauthorized');
  }

  // Attach user object to request for route handlers to use
  req.user = user;
  next();
};

// Route that requires authentication
app.get('/profile/:userId', authenticate, (req, res) => {
  const user = req.user;
  res.send(`Welcome, ${user.username}! This is your profile.`);
});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Use Cases of Middleware:

- Middleware serves various purposes in web applications:
- Authentication and Authorization: Middleware can check if a user is authenticated or has the necessary permissions to access a particular route.
- Logging: Middleware can log information about incoming requests, helping with debugging and tracking user behavior.
- Request Body Parsing: Middleware can parse the incoming request body (e.g., JSON, form data) and attach it to the req object for easier handling in route handlers.
- Error Handling: Middleware can catch errors that occur during request processing and provide an appropriate response.
- CORS (Cross-Origin Resource Sharing): Middleware can set appropriate headers to enable CORS and control which origins are allowed to access the server.

- Compression: Middleware can compress response data to improve performance.

Middleware Execution Order:
Middleware functions are executed in the order they are added to the application. The order matters, as each middleware can modify the req and res objects, and subsequent middleware functions or route handlers will work with the modified objects.

What is params?
"params" (short for "parameters") refers to the dynamic segments in the URL path. These segments can contain variable data that is extracted from the URL when a client makes a request. Parameters are used to pass values to a route handler in order to customize the behavior of that handler based on the data in the URL.
Express.js allows you to define routes with parameters and access those parameters within your route handlers. This is particularly useful when you have routes that need to handle different data based on the URL.

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

// Route with a parameter
app.get('/user/:userId', (req, res) => {
  const userId = req.params.userId; // Access the parameter using req.params
  res.send(`User ID: ${userId}`);
});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Route Handling Methods:
Route handling methods in Express.js are used to define how your application should respond to different HTTP methods and specific routes. These methods allow you to set up routes for different URL paths and HTTP verbs (GET, POST, PUT, DELETE, etc.). Each route handling method corresponds to a specific HTTP method and defines what should happen when a request matches that route.

1. app.get(path, callback) - Handling GET Requests:
   This method defines a route for handling HTTP GET requests with the specified path. When a client makes a GET request to the specified path, the provided callback function is executed. The callback function receives two objects: req (request) and res (response), which you can use to interact with the incoming request and send a response back to the client.

2. app.post(path, callback) - Handling POST Requests:
   Similar to app.get(), this method defines a route for handling HTTP POST requests with the specified path. The provided callback function is executed when a client sends a POST request to the specified path. You can use this to process form data, create new resources, or perform other actions based on the incoming data.
3. app.put(path, callback) - Handling PUT Requests:
   This method defines a route for handling HTTP PUT requests with the specified path. When a client sends a PUT request to the defined path, the callback function is executed. PUT requests are often used to update existing resources, and the data to be updated is usually included in the request body.
4. app.delete(path, callback) - Handling DELETE Requests:
   This method defines a route for handling HTTP DELETE requests with the specified path. When a client sends a DELETE request to the defined path, the callback function is executed. DELETE requests are used to remove resources, and the resource to be deleted is often identified by parameters in the request.
5. app.all(path, callback) - Handling All HTTP Methods:
   app.all() defines a route that matches any HTTP method (GET, POST, PUT, DELETE, etc.) for the specified path. The provided callback function is executed for all incoming requests to the defined path, regardless of the HTTP method used. This can be useful for applying general actions or validation to multiple methods for a specific path.
6. app.use(path, callback) - Middleware for All Routes:
   app.use() is used to apply middleware functions to one or more routes. Middleware functions are functions that run before the final route handler and have access to the req and res objects. This method is often used for tasks like parsing request bodies, authentication, logging, and more. The middleware function can be applied globally to all routes or to specific routes.

> ! action in form tells which endpoint we need to hit when we submit the form. By default the action will redirect it in the same file in which form is created.
> ! method in form tells which routing handling method it will allow when we click on submit button. By default the method is get.

what is urlencoded??

what is mongoDB and why do we use it?
MongoDB is a popular open-source, NoSQL (non-relational) database management system. It falls under the category of document-oriented databases, which means it stores and retrieves data in a format similar to JSON documents. MongoDB is designed to handle large volumes of unstructured or semi-structured data.
Key features of MongoDB include:

Schema Flexibility: Unlike traditional relational databases, MongoDB does not require a predefined schema. Documents within a collection can have different structures, allowing for more flexible data modeling.
Rich Query Language: MongoDB provides a powerful query language with support for filtering, sorting, projection, and aggregation operations, enabling complex data retrieval and analysis.
Replication and High Availability: MongoDB supports data replication across multiple nodes, ensuring high availability and fault tolerance. It can automatically handle failover in case a primary node becomes unavailable.
Scaling: MongoDB can be scaled both horizontally (across multiple machines) and vertically (within a single machine) to handle large amounts of data and traffic.
JSON-Like Documents: Since MongoDB stores data in a JSON-like format, it aligns well with modern programming languages and web technologies.This includes the use of key-value pairs, nested objects and arrays.

why mongodb uses json like structures to store data?
MongoDB uses JSON-like structures to store data because it aligns well with the dynamic and flexible nature of the NoSQL document-oriented database model. JSON-like structures provide several benefits that make them suitable for storing data in MongoDB:
Flexibility: JSON-like structures allow documents to have varying structures within the same collection. This means that different documents can have different fields, data types, and nesting levels, without the need for a predefined schema. This flexibility is particularly useful in scenarios where data models evolve over time.
Schema-less Design: MongoDB's use of JSON-like structures supports a schema-less design, allowing developers to easily add, modify, or remove fields from documents without needing to alter a centralized schema definition. This makes it simpler to adapt to changing application requirements.
Readability and Ease of Use: JSON-like structures are human-readable and resemble the way data is represented in many programming languages. This makes it easier for developers to work with data and understand its structure.
Integration with Programming Languages: JSON-like structures can be directly used in many programming languages, simplifying data manipulation and serialization/deserialization tasks. This alignment with programming languages makes it seamless to work with MongoDB in various application contexts.
Simplicity of Representation: JSON-like structures do not require complex encoding or decoding processes. JSON data can be directly serialized to and deserialized from BSON (Binary JSON), which is the binary representation used by MongoDB, making data storage and retrieval efficient.
Rich Data Modeling: JSON-like structures support nested objects and arrays, enabling developers to represent complex relationships and hierarchical data models without requiring multiple tables or collections as in relational databases.

Schema Validation: While MongoDB supports flexible data models, it also offers optional schema validation to ensure that certain fields or data types conform to a predefined structure, providing a balance between flexibility and data integrity.
Simpler Migration and Development: The JSON-like nature of MongoDB makes it easier to migrate data from other JSON-based sources and to develop applications that interact with the database, reducing the impedance mismatch between application and data store.

process.nextTick(): Callbacks scheduled with this method are executed immediately after the current operation completes, before any I/O events or timers are triggered. It's placed in front of the event loop queue. This is useful for tasks that need to be performed before I/O events.
setImmediate(): Callbacks scheduled with this method are executed on the next iteration of the event loop after I/O events, allowing I/O events to be processed between each callback execution. It's placed at the end of the event loop queue. It's used to ensure that code runs after the I/O events of the current iteration.
Both methods allow you to control the order of execution within the event loop, but they have different behaviors regarding when they execute relative to I/O events and timers.

eventEmitter???

package.json - It is a core component of Node.js projects and serves as a configuration and metadata file for your application. It contains various information about your project, its dependencies, scripts, project details and other important settings.

> The URL module in Node.js provides various utilities for URL resolution and parsing. It is a built-in module that helps split up the web address into a readable format. url.resolve(), url.format().

> Streams in Node.js are a fundamental concept used for handling and manipulating data, especially when working with large amounts of data or in situations where data needs to be processed as it's being read or written, rather than loading it all into memory at once.

RELP???

Control flow in Node.js manages function calls by determining the order in which asynchronous operations are executed and how the program flows through different parts of the code.
It can be done using callbacks, promises and async/await.

Can you access DOM in Node?
No, you cannot access the DOM in Node.js. The DOM is a browser-specific API that allows for the manipulation of HTML and XML documents. Since Node.js does not run in a browser, it does not have access to the DOM.

fork() and spawn() in node js.
buffer class in node js.
piping in node js.
callback hell?
NODE_ENV?
Test Pyramid in node js?
event emitter in node js?