Nodejs #nodejs

How Node JS works
Callbacks, Callback hell, Promise, Async Await
All ES6 Concepts
Closure
Lexical Scope
Var, let, const
Hoisting
Scope chain
Higher order functions
Need of express js
Middleware
Inversion Control
Routing
JWT
Protected Route
Error Handling
Writing Test Cases in Node js
Typescript
Security
Multithreading (exec, fork, spawn)
Microservices

1. Primitive Data Types:
Primitive data types are the basic building blocks of data representation. They are directly supported by the programming language and are not composed of other data types. Examples of primitive data types include:

- **Integer:**
  - Examples: int, short, long
- **Floating-Point:**
  - Examples: float, double
- **Character:**
  - Examples: char
- **Boolean:**
  - Examples: boolean
- **String (In some languages, considered a primitive type):**
  - Examples: string

2. Non-Primitive (Composite) Data Types:
Non-primitive data types are more complex and are composed of multiple primitive data types or other non-primitive data types. They are often referred to as composite or structured data types. Examples of non-primitive data types include:

- **Arrays:**
  - Ordered collection of elements of the same data type.

- **Structures (or Records):**
  - Collection of elements, each with its own data type.
- **Classes (Object-Oriented Programming):**
  - Blueprint for creating objects, encapsulating both data and methods.
- **Pointers:**
  - Variables that store memory addresses.
- **Lists:**
  - Dynamic data structures that can grow or shrink during program execution.
- **Sets:**
  - Unordered collection of unique elements.
- **Maps (or Dictionaries):**
  - Key-value pairs, where each key is associated with a value.
- **Files:**
  - Representations of files, often used for input/output operations.

Explain the difference between synchronous and asynchronous code in Node.js.

Synchronous code blocks the execution until it has finished running, while asynchronous code does not. Asynchronous code allows for non-blocking I/O operations, meaning that it can run concurrently with other code. In Node.js, most of the built-in modules are asynchronous.

```
// Synchronous code
const fs = require('fs');
const data = fs.readFileSync('file.txt');
console.log(data);

// Asynchronous code with callbacks
const fs = require('fs');
fs.readFile('file.txt', (err, data) => {
  if (err) {
    console.error(err);
  } else {
    console.log(data);
  }
});

// Asynchronous code with promises
```

```
const fs = require('fs').promises;
fs.readFile('file.txt')
  .then(data => console.log(data))
  .catch(error => console.error(error));

// Asynchronous code with async/await
const fs = require('fs').promises;
async function readFile() {
  try {
    const data = await fs.readFile('file.txt');
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
readFile();
```

What is the difference between EventEmitter and streams in Node.js?

EventEmitter is a class that allows developers to implement and handle custom events in their applications. It allows for communication between different parts of the application. Streams, on the other hand, are used for handling continuous streams of data in Node.js, such as reading or writing large files. They can be used to read data piece-by-piece as it becomes available, without having to wait for the entire data set to be loaded.

```
const { EventEmitter } = require('events');
const { Readable } = require('stream');

// Example 1: EventEmitter
const eventEmitter = new EventEmitter();

eventEmitter.on('message', (message) => {
  console.log(`Received message: ${message}`);
});

eventEmitter.emit('message', 'Hello, world!');

// Example 2: Readable stream
const readable = new Readable({
  read() {}
});

readable.push('Hello, ');
```

```
readable.push('world!');
readable.push(null);

readable.pipe(process.stdout);
```

What are the best practices for error handling in Node.js?

Some best practices for error handling in Node.js include:

- Always handle errors with try-catch statements or callbacks.
- Use error-first callbacks.
- Use descriptive error messages.
- Log errors and relevant information.
- Don't ignore errors.
- Use middleware to handle errors.

```
// Error handling with try/catch

try {
  const data = fs.readFileSync('file.txt');
  console.log(data);
} catch (error) {
  console.error(error);
}
```

```
// Use error-first callbacks.
```

In Node.js, error-first callbacks are a common pattern used to handle errors in asynchronous code. The basic idea is to pass a callback function to an asynchronous function as the last argument, and that callback function should have an error object as its first argument. If an error occurs during the execution of the asynchronous function, the error object should be populated and passed as the first argument to the callback function. If no error occurs, the error object should be null or undefined.
Here is an example of using an error-first callback to read a file asynchronously with the fs module:
javascript

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

```
fs.readFile('file.txt', (err, data) => {
  if (err) {
    console.error('Error:', err);
    return;
  }

  console.log('Data:', data);
});
```

Middleware is a powerful concept in Node.js that allows you to add functionality to your application's request/response processing pipeline. Middleware functions are simply functions that take three arguments: the request object, the response object, and a next function that is used to pass control to the next middleware function in the pipeline.
One common use case for middleware is to handle errors in a centralized and consistent way across your application. You can define an error-handling middleware function that will be called if any previous middleware function in the pipeline throws an error.
Here is an example of using middleware to handle errors in a Node.js application:

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

app.use((req, res, next) => {
  try {
    // Call next middleware function
    next();
  } catch (err) {
    // Handle error
    res.status(500).send('Internal Server Error');
  }
});

app.get('/', (req, res) => {
  throw new Error('Oops!');
});

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

In this example, the error-handling middleware function is defined using app.use(). The function calls next() to pass control to the next middleware

function in the pipeline. If an error occurs in any previous middleware function, the error-handling middleware function will be called and the error will be handled.
In this case, the error-handling middleware function sends a 500 Internal Server Error response to the client with the message "Internal Server Error". By using middleware to handle errors, you can ensure that your application responds consistently to errors and provides a good user experience even in the case of unexpected errors.

Child process in NodeJs ?

In Node.js, a child process is a separate Node.js process that can be executed from the main Node.js process. This allows for running external scripts or commands from within a Node.js application.
Node.js provides a built-in module called child_process that makes it easy to create and manage child processes. The module provides several functions, including spawn, exec, execFile, and fork, each of which has its own use case and benefits.

Spawn	**Fork**
Designed to run system commands.	A special instance of spawn() that runs a new instance of V8.
Does not execute any other code within the node process.	Can create multiple workers that run on the same Node codebase.
child_process.spawn(command[, args][, options]) creates a new process with the given command.	Special case of spawn() to create child processes using. child_process.fork(modulePath[, args][, options])
Creates a streaming interface (data buffering in binary format) between parent and child process.	Creates a communication (messaging) channel between parent and child process.
More useful for continuous operations like data streaming (read/write). For example, streaming images/files from the spawn process to the parent process.	More useful for messaging. For example, JSON or XML data messaging.

The spawn function is the most commonly used function for creating child

processes. It launches a command in a new process and provides a stream for reading and writing to the process.

The exec function is similar to spawn, but it buffers the output of the child process into memory and provides the output in a callback.

The execFile function is similar to exec, but it allows you to specify the command to be executed as a file path instead of a command string.

Finally, the fork function is a special case of spawn that is specifically designed for creating child processes that communicate with each other using a messaging system.

In summary, child processes in Node.js provide a powerful way to execute external commands and scripts from within a Node.js application. They can be created using the child_process module, and there are several functions available for different use cases.

One real-life example of using child_process in Node.js is when a web server needs to run a command-line tool or script that performs some task, such as image processing or data manipulation.

For example, let's say you have a Node.js web application that needs to resize a large number of images on the server. Rather than writing a custom image processing module in Node.js, you could use an existing command-line tool like ImageMagick, which provides a suite of powerful image manipulation tools.

You could use the child_process module in Node.js to spawn a child process that runs the ImageMagick command-line tool, passing it the necessary arguments to resize the images. The child process could then communicate back to the parent process with the results of the image processing, which could be displayed on the web page or stored in a database.

Here is an example code snippet that demonstrates how you could use the child_process module to spawn a child process that runs the ImageMagick "convert" command to resize an image:

```javascript
const { spawn } = require('child_process');

const convert = spawn('convert', ['input.jpg', '-resize', '50%', 'output.jpg']);

convert.stdout.on('data', (data) => {
  console.log(`stdout: ${data}`);
});
```

```
convert.stderr.on('data', (data) => {
  console.error(`stderr: ${data}`);
});

convert.on('close', (code) => {
  console.log(`child process exited with code ${code}`);
});
```

In this example, the `spawn` function is used to spawn a child process that runs the "convert" command with the given arguments. The `stdout`, `stderr`, and `close` events are used to handle output and errors from the child process, and to handle when the child process exits.

WHAT ARE THE PHASES OF EVENT LOOOOOP ?

The event loop is a critical component of Node.js that enables it to handle I/O operations in an efficient, non-blocking manner. Here are the different phases of the event loop in Node.js:

1. Timers: In this phase, the event loop checks if there are any scheduled timers that have expired. If there are, the corresponding callback functions are added to the callback queue and will be executed in the next phase of the event loop.

2. I/O callbacks: In this phase, any I/O callback functions that were deferred from the previous event loop iteration are processed. This includes callback functions for network operations, file system operations, and other types of I/O.

3. Idle, prepare: These are internal phases of the event loop that are used to perform any necessary setup or cleanup operations between event loop iterations.

4. Poll: In this phase, the event loop waits for new events to arrive. This can include new I/O events, timers, or other types of events. If there are no new events, the event loop will wait until a new event arrives or until a timer expires.

5. Check: In this phase, any setImmediate() callback functions that were scheduled during the current event loop iteration are executed.

6. Close callbacks: In this phase, any 'close' event callback functions are executed. These are typically used to perform cleanup operations after a connection or resource is closed.

After completing all of these phases, the event loop will start over at the beginning with the Timers phase. The event loop continues to run as long as there are any active timers or pending I/O operations.

WHEN TO USE PROMISES AND ASYNC AWAIT ?

Both Promises and `async/await` are features in JavaScript used to handle asynchronous code execution.

A Promise is an object that represents a value that may not be available yet but will be resolved at some point in the future. Promises can be used to handle asynchronous operations by registering callbacks to handle the resolution of the Promise.

`async/await` is a syntax feature in JavaScript that makes asynchronous code look more like synchronous code. It allows you to write asynchronous code that looks and behaves like synchronous code, making it easier to read and understand.

Here are some key differences between Promises and `async/await`:

Syntax	With Promises, you chain `.then()` and `.catch()` methods to handle the resolution or rejection of a Promise.	With `async/await`, you use the `async` keyword to define a function as asynchronous, and use the `await` keyword to wait for a Promise to resolve before continuing execution.
Error handling	With Promises, you can use the `.catch()` method to handle errors.	With `async/await`, you can use a `try/catch` block to handle errors.

Readability	NONE	`async/await` is generally considered to be more readable and easier to follow than Promise chains, as it allows you to write asynchronous code that looks more like synchronous code

When to use Promises:

Promises are a good choice when you need to perform a sequence of asynchronous operations that depend on the results of previous operations, or when you need to handle errors and control the flow of execution based on the results of asynchronous operations.

When to use `async/await`:

`async/await` is a good choice when you need to write asynchronous code that is more readable and easier to follow, especially when you have a series of operations that need to be performed in order. `async/await` can also make error handling easier, as you can use `try/catch` blocks to catch errors instead of using `.catch()` methods.

Difference between async.parallel and async.series ?

`async.parallel` and `async.series` are two methods in the Async.js library that allow for executing multiple asynchronous tasks in a specific order.

The main difference between them is how they handle the asynchronous tasks.

- `async.parallel`: The `async.parallel` method executes multiple asynchronous tasks simultaneously and then invokes a callback function once all of them are completed. It takes an array of functions or an object where each property is a function, and each function executes a specific task. This method returns an array of results from each function in the same order as provided in the input.

- `async.series`: The `async.series` method executes multiple asynchronous tasks in a sequential order and then invokes a callback function once all of them are completed. It takes an array of functions

or an object where each property is a function, and each function executes a specific task. This method returns an array of results from each function in the same order as provided in the input.

So, the main difference between them is that `async.parallel` executes all tasks at the same time and returns an array of results while `async.series` executes tasks sequentially and returns an array of results in the same order as tasks were provided.

Difference between promise.all and promise.allsettled ?

`Promise.all` and `Promise.allSettled` are two methods in JavaScript that allow you to handle multiple promises simultaneously.

The main difference between them is in how they handle the returned values and errors from the promises.

- `Promise.all`: The `Promise.all` method takes an array of promises and returns a single promise. This single promise resolves with an array of resolved values from the input promises, in the same order as the input promises. If any of the input promises is rejected, the returned promise is immediately rejected with the reason of the first rejected promise. This means that if any promise in the input array is rejected, the entire operation is considered a failure.

- `Promise.allSettled`: The `Promise.allSettled` method also takes an array of promises and returns a single promise. This single promise resolves with an array of objects, one for each input promise. Each object has two properties: `status`, which is either `"fulfilled"` or `"rejected"`, and `value` or `reason`, which is the value returned by the resolved promise or the reason for the rejected promise, respectively. This means that even if some of the input promises are rejected, the returned promise will still resolve with an array of all results, both fulfilled and rejected.

In summary, `Promise.all` fails fast and rejects immediately if any of the input promises is rejected, while `Promise.allSettled` always resolves with an array of objects that describe the status of all input promises, regardless of whether they were fulfilled or rejected.

What is control function ?

A control function manages and manipulates the flow of asynchronous code

execution.
Node.js is designed to handle asynchronous I/O operations, which means that multiple I/O operations can be executed simultaneously without blocking the execution of other code. However, managing the flow of asynchronous code can be challenging, especially when multiple operations need to be executed in a particular order.
Control functions provide a solution to this problem by allowing developers to define the order in which asynchronous operations should be executed. They can be used to perform tasks such as error handling, callback management, and flow control.

How to handle uncatchable exceptions in nodejs ?

In Node.js, if an uncaught exception occurs, the process will terminate by default. However, there are a few ways to handle uncatchable exceptions:

1. `process.on('uncaughtException', callback)`: This method can be used to add a listener for the `uncaughtException` event, which will be emitted when an uncaught exception is thrown. This allows you to perform cleanup tasks and prevent the process from exiting.

2. `process.on('unhandledRejection', callback)`: This method can be used to add a listener for the `unhandledRejection` event, which will be emitted when a Promise is rejected but no error handler is attached. This allows you to catch unhandled Promise rejections and perform appropriate error handling.

3. `domain`: A deprecated module in Node.js that provides a way to handle uncaught exceptions in a more granular way. However, it is not recommended to use it in new applications as it has been deprecated.

It is important to note that while these methods can handle uncatchable exceptions, it is still best practice to catch exceptions wherever possible to prevent unexpected behaviour and ensure the stability of your application.

Uncatchable exceptions are typically exceptions that occur outside of a `try...catch` block or when a Promise is rejected but no error handler is attached. Here is an example of an uncatchable exception:

```js
const fs = require('fs');

fs.readFile('non-existent-file.txt', (err, data) => {
  if (err) {
```

```
    throw new Error('Unable to read file'); // uncaught exception
  }
  console.log(data);
});
```

In this example, the `readFile()` method will throw an error because the file `non-existent-file.txt` does not exist. Since the error is thrown outside of a `try...catch` block, it becomes an uncaught exception, and the process will terminate. To handle this exception, you can use the `process.on('uncaughtException', callback)` method to add a listener for the `uncaughtException` event and perform appropriate error handling:

```js
process.on('uncaughtException', (err) => {
  console.error('An uncaught exception occurred:', err);
  // perform cleanup tasks
  process.exit(1);
});

const fs = require('fs');

fs.readFile('non-existent-file.txt', (err, data) => {
  if (err) {
    throw new Error('Unable to read file');
  }
  console.log(data);
});
```

In this modified example, the `process.on('uncaughtException', callback)` method is used to add a listener for the `uncaughtException` event. The callback function will be executed when an uncaught exception occurs, and it logs the error message and exits the process with a non-zero exit code to indicate that an error occurred.

Explain the concept of backpressure in Node.js streams and why it is important to handle it properly. ?

Backpressure refers to a situation where the data producer generates data faster than the data consumer can process it. In the context of Node.js streams, this often occurs when a fast Readable stream (data producer) is piped to a slower Writable stream (data consumer).
Handling backpressure is crucial to prevent the buffering of an excessive

amount of unprocessed data in memory, which can impact the application's performance or even crash it.
Node.js streams are designed to handle backpressure automatically when using the built-in .pipe() method, adjusting the producer's data generation rate to match the consumer's processing rate. However, if you implement custom data handling or flow control mechanisms, you may need to account for backpressure management manually. Key principles to handle backpressure effectively include:

- Pausing the Readable stream when the Writable stream's buffer exceeds its high watermark threshold.
- Resuming the Readable stream only after the Writable stream's buffer has been drained below the low watermark threshold.
- Utilizing available events and methods provided by the stream API, such as 'drain', 'readable', 'write', and 'end'.

By properly handling backpressure, you can maintain a well-managed flow of data through your application, improving its overall performance and stability.

What is a first-class function in Javascript?

In JavaScript, functions are considered first-class citizens, which means they are treated like any other value, such as numbers, strings, or objects. This concept of "first-class functions" has several implications and features in the JavaScript programming language:

1. **Functions as Values:** In JavaScript, functions can be assigned to variables, passed as arguments to other functions, returned from functions, and stored in data structures like arrays and objects. This flexibility allows you to work with functions in a similar way as you work with other types of data.

2. **Function Expressions:** You can define functions using function expressions, which involve assigning an anonymous function to a variable. This variable can then be used to call the function.

```javascript
const greet = function(name) {
  return `Hello, ${name}!`;
};

console.log(greet("Alice")); // Output: Hello, Alice!
```

3. **Higher-Order Functions:** Functions that accept other functions as

arguments or return functions are called higher-order functions. This pattern is central to many functional programming concepts and allows for elegant and concise code.

```javascript
function operate(func, a, b) {
  return func(a, b);
}

function add(x, y) {
  return x + y;
}

function multiply(x, y) {
  return x * y;
}

console.log(operate(add, 3, 5));      // Output: 8
console.log(operate(multiply, 2, 4)); // Output: 8
```

4. **Returning Functions:** Functions can return other functions as their result. This is known as "function returning a function" or "closure" behavior.

```javascript
function multiplier(factor) {
  return function(x) {
    return x * factor;
  };
}

const double = multiplier(2);
console.log(double(5)); // Output: 10
```

5. **Anonymous Functions:** You can create anonymous functions (functions without a name) and use them directly where needed.

```javascript
const numbers = [1, 2, 3, 4, 5];
const squared = numbers.map(function(x) {
  return x * x;
});

console.log(squared); // Output: [1, 4, 9, 16, 25]
```

Overall, the concept of first-class functions in JavaScript enables you to write more modular, flexible, and expressive code by treating functions as versatile and reusable entities.

Performance Optimization Techniques

In the realm of Node.js development, performance optimization is paramount. Ensuring that applications run efficiently can lead to better user experiences, reduced server costs, and overall improved application reliability. Here are some techniques to enhance the performance of your Node.js applications.

Profiling And Monitoring
Optimize Database Queries
Use Caching
Use Compression
Avoid Synchronous Code

Profiling And Monitoring

Regularly **profiling** and monitoring your application can help identify bottlenecks. Tools like Node Clinic or N|Solid provide insights into the runtime performance and help pinpoint areas of improvement.
// Using Node's built-in profiler
node --prof app.js

After running your application with the --prof flag, Node.js will generate a .log file. Analyzing this file can provide valuable insights into your application's performance.

Security Considerations In Node.js
Security is paramount in any application, and Node.js is no exception. Ensuring that your Node.js applications are secure can protect sensitive data, maintain user trust, and prevent malicious attacks. Here are some key security considerations to keep in mind.

Dependency Management
Input Validation
Use HTTPS
Avoid Using Eval()
Rate Limiting

Dependency Management

Regularly updating your **dependencies** can patch known vulnerabilities. Tools

like npm audit can help identify and fix potential security issues in your packages.
npm audit
Running the above command will check your project for known vulnerabilities and provide recommendations on how to address them.

Input Validation

Always validate and sanitize **user input** to prevent attacks like SQL injection or cross-site scripting (XSS). Libraries like validator or express-validator can assist in this process.

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

// Validate email
const email = 'test@example.com';
if (validator.isEmail(email)) {
    // Process email
} else {
    console.error('Invalid email');
}
```

This example demonstrates how to validate an email address using the validator library.

Use HTTPS

Always use **HTTPS** for transmitting data to ensure that it's encrypted and secure. This is especially crucial for sensitive data like passwords or payment details.

```
const https = require('https');
const fs = require('fs');

const options = {
    key: fs.readFileSync('key.pem'),
    cert: fs.readFileSync('cert.pem')
};

https.createServer(options, (req, res) => {
    res.writeHead(200);
    res.end('Secure content');
}).listen(8080);
```

This code sets up a basic HTTPS server using Node.js, ensuring encrypted communication.

Avoid Using Eval()

The **eval()** function can execute arbitrary code, making it a potential security risk. Avoid using it to prevent code injection attacks.

```
// Avoid this
eval('console.log("Hello World!")');

// Prefer safer alternatives
const log = Function('return console.log("Hello World!")');
log();
```

By using the Function constructor, you can achieve similar functionality without the risks associated with eval().

Rate Limiting

Implement **rate limiting** to prevent brute-force attacks or denial-of-service attacks. Libraries like express-rate-limit can help set up rate limiting for your applications.

```
const rateLimit = require('express-rate-limit');
const express = require('express');
const app = express();

const limiter = rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 100 // limit each IP to 100 requests per windowMs
});

app.use(limiter);
```

This example sets up rate limiting for an Express application, limiting each IP to 100 requests every 15 minutes.
Security should always be a top priority in application development. By following these considerations and staying updated on best practices, developers can ensure that their Node.js applications remain secure and resilient against potential threats.

If Node.js is single threaded then how does it handle concurrency?

The main loop is single-threaded and all async calls are managed by libuv library.

For example:

```
const crypto = require("crypto");
const start = Date.now();
function logHashTime() {
 crypto.pbkdf2("a", "b", 100000, 512, "sha512", () => {
console.log("Hash: ", Date.now() - start);
 });
}
```

```
logHashTime();
logHashTime();
logHashTime();
logHashTime();
```

This gives the output:

```
Hash: 1213
Hash: 1225
Hash: 1212
Hash: 1222
```

This is because libuv sets up a thread pool to handle such concurrency. How many threads will be there in the thread pool depends upon the number of cores but you can override this.

Differentiate between process.nextTick() and setImmediate()?

Both process.nextTick() and setImmediate() are part of the Node.js event loop and are used for scheduling asynchronous code to be executed in the future. However, they have different behaviours and priorities within the event loop:

- process.nextTick():
  - process.nextTick() is used to defer the execution of a callback function until the next iteration of the event loop.
  - Callbacks scheduled with process.nextTick() are executed before any other callbacks in the current iteration of the event loop, including I/O operations, timers, and other scheduled tasks.
  - This means that if you have multiple process.nextTick() callbacks, they will be executed one after another in the order they were added, and other event loop tasks will be delayed.
  - It's a mechanism for placing a function at the front of the event queue, ensuring it will execute before other I/O events.
  - Because of its immediate priority, excessive use of process.nextTick() can potentially block the event loop and cause starvation of other I/O operations.

- setImmediate():
  - setImmediate() is used to schedule a callback function to be executed on the next iteration of the event loop, after I/O events and any pending timers.
  - Callbacks scheduled with setImmediate() are executed after any process.nextTick() callbacks but before timers.
  - It provides a way to defer a function's execution to a later iteration of the event loop without blocking other I/O tasks.
  - It's generally used when you want to ensure that your callback is

executed in a more balanced manner with respect to other I/O events.
    - It's particularly useful when dealing with I/O-bound tasks or when you want to break up a long-running task to avoid blocking the event loop.

Both can be used to switch to an asynchronous mode of operation by listener functions.

process.nextTick() sets the callback to execute but setImmediate pushes the callback in the queue to be executed. So the event loop runs in the following manner

timers–>pending callbacks–>idle,prepare–>connections(poll,data,etc)–>check–>close callbacks
In this process.nextTick() method adds the callback function to the start of the next event queue and setImmediate() method to place the function in the check phase of the next event queue.

How does Node.js overcome the problem of blocking of I/O operations?

Since the node has an event loop that can be used to handle all the I/O operations in an asynchronous manner without blocking the main function.

So for example, if some network call needs to happen it will be scheduled in the event loop instead of the main thread(single thread). And if there are multiple such I/O calls each one will be queued accordingly to be executed separately(other than the main thread).
Thus even though we have single-threaded JS, I/O ops are handled in a nonblocking way.

What's the difference between a web worker and a worker thread?

Web workers are implemented in the browser and worker threads are implemented in Node.js. They both resolve the same issue, which is to provide parallel processing. In fact, the Worker Thread API is based on the Web Workers implementation.

What are the advantages of using a worker thread vs a child process?

While a child process runs its own process with its own memory space, a

worker thread is a thread within a process that can share memory with the main thread. This helps to avoid expensive data serializations back and forth.

Consider following code snippet:

```
{
   console.time("loop");
   for (var i = 0; i < 1000000; i += 1){
      // Do nothing
   }
   console.timeEnd("loop");
}
```

The time required to run this code in Google Chrome is considerably more than the time required to run it in Node.js. Explain why this is so, even though both use the v8 JavaScript Engine.

Within a web browser such as Chrome, declaring the variable i outside of any function's scope makes it global and therefore binds it as a property of the window object. As a result, running this code in a web browser requires repeatedly resolving the property i within the heavily populated window namespace in each iteration of the for loop.
In Node.js, however, declaring any variable outside of any function's scope binds it only to the module's own scope (not the window object) which therefore makes it much easier and faster to resolve.
It's also worth noting that using let instead of var in the for loop declaration can reduce the loop's run time by over 50%. But such a change assumes you know the difference between let and var and whether this will have an effect on the behavior of your specific loop.

Explain the purpose of the require function in Node.js and how it facilitates module loading. How do you load core modules, built-in modules, and local modules using require?

Answer:
The require function in Node.js is used to load modules and files. It allows you to include external modules, built-in modules, and custom modules in your Node.js application.
To load core modules or built-in modules provided by Node.js, you can use the require function directly:

```
const http = require('http');
const fs = require('fs');
const path = require('path');
```
Code language: JavaScript (javascript)

Core modules are modules that are included by default in Node.js, and you can use them without installing any additional packages.

To load local modules or custom modules created by you or other developers, you need to provide the file path relative to the current file:

Assume you have a file named mathUtils.js in the same directory as your current file:

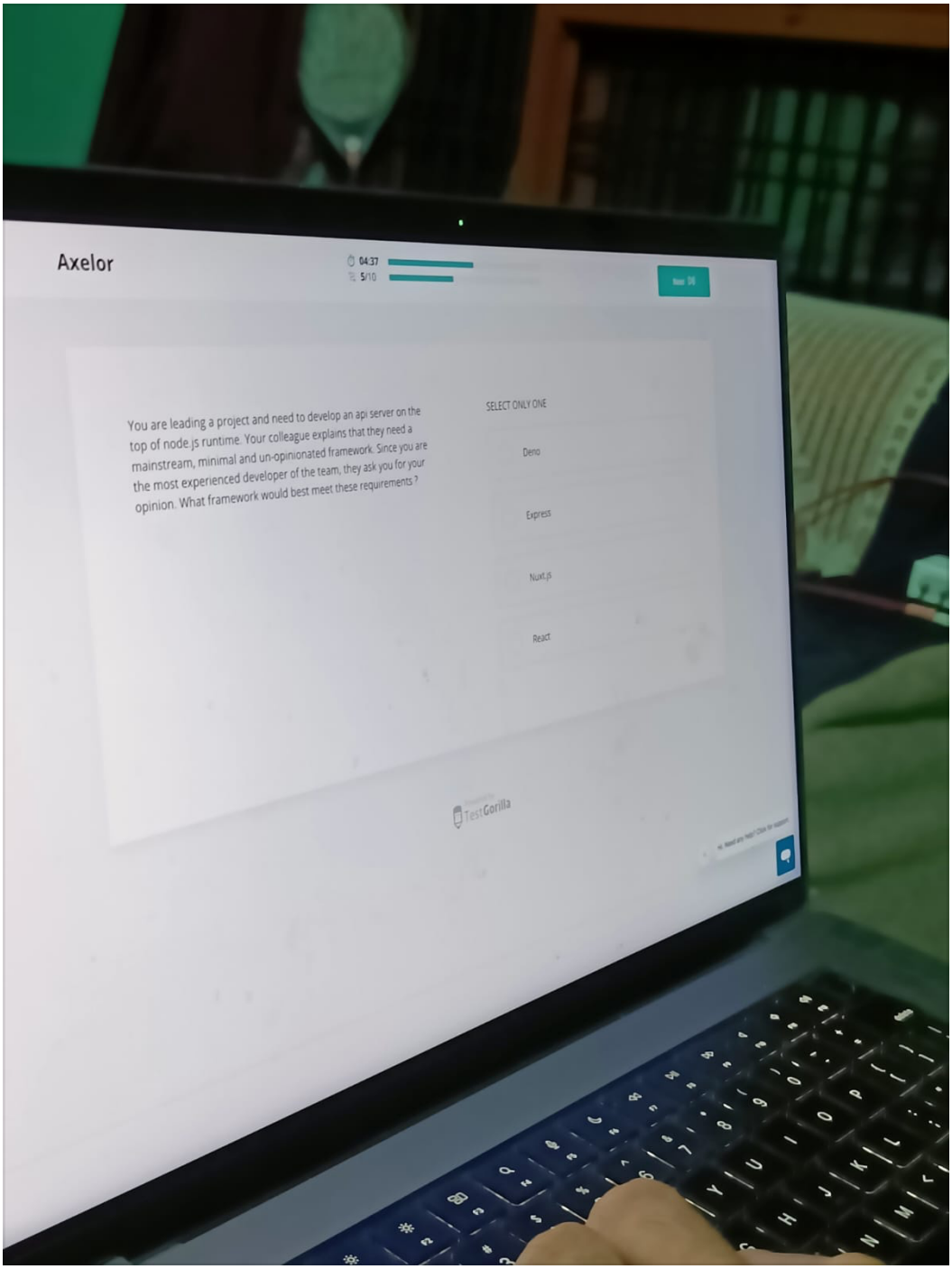
```
const mathUtils = require('./mathUtils');
```
Code language: JavaScript (javascript)

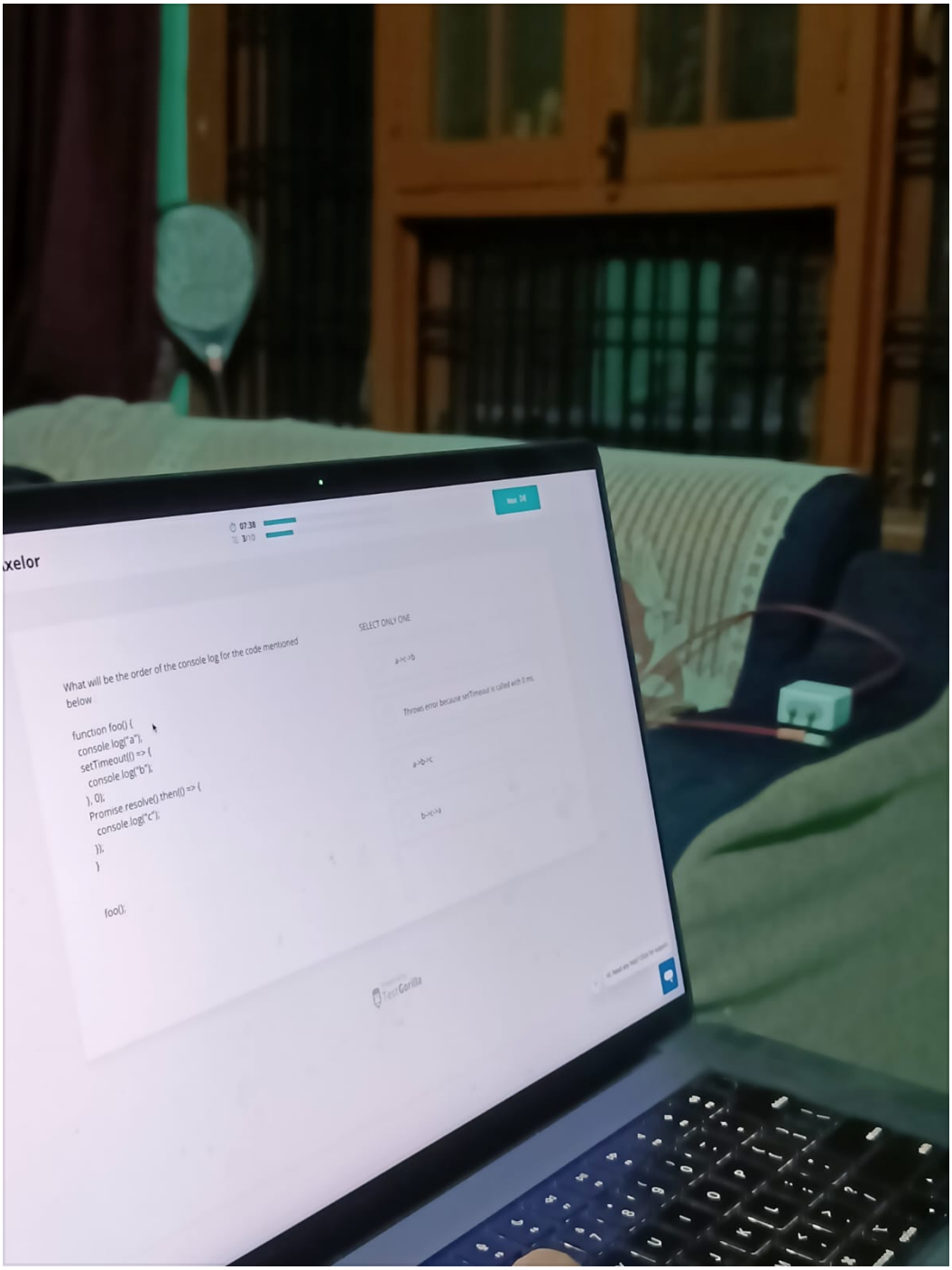
In this example, we load the custom module mathUtils.js using require. The ./ indicates that the module is in the same directory as the current file.

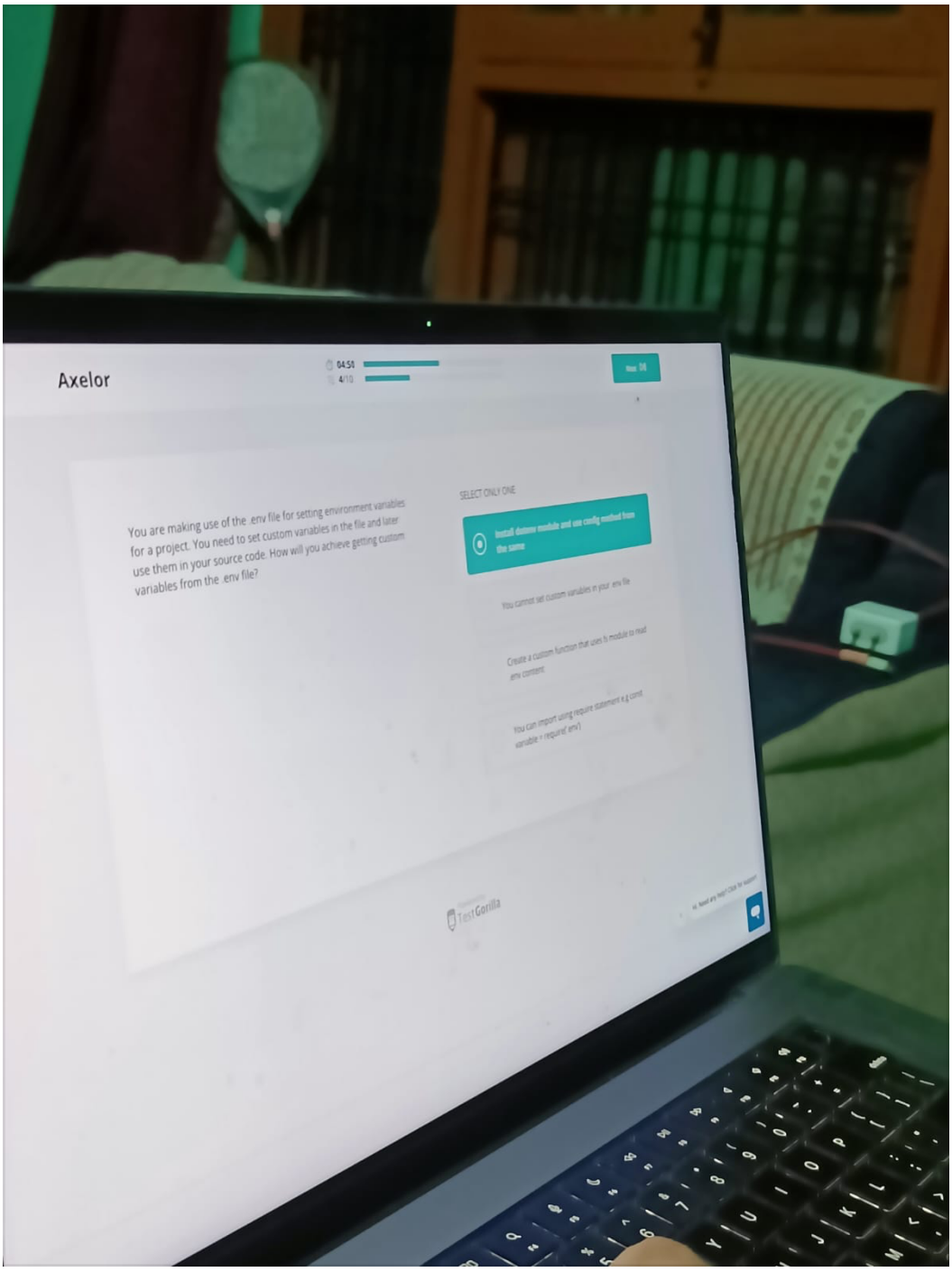
Question solve this ....

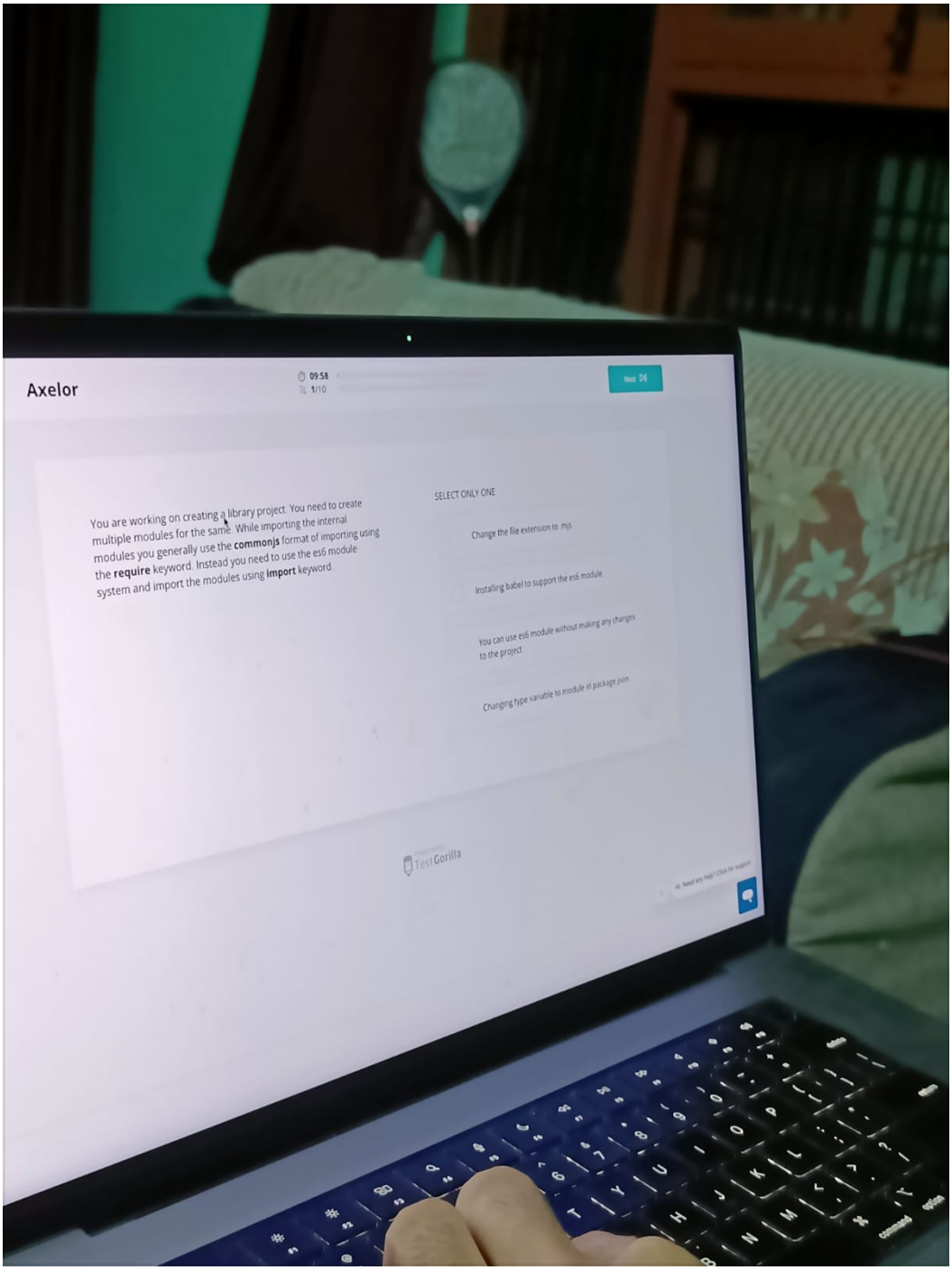
```
console.log("Hi");

// First timeout
const timeout1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    console.log("wait for sec");
    const success = true;
    if (success) {
      resolve("Success!");
    } else {
      reject("Error!");
    }
  }, 1000);
});

console.log("Hi2");

// Second timeout
const timeout2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    console.log("wait");
    resolve("Done!");
  }, 0);
});

// Execution after both timeouts
Promise.resolve().then(() => {
  console.log("Promise resolved");
```

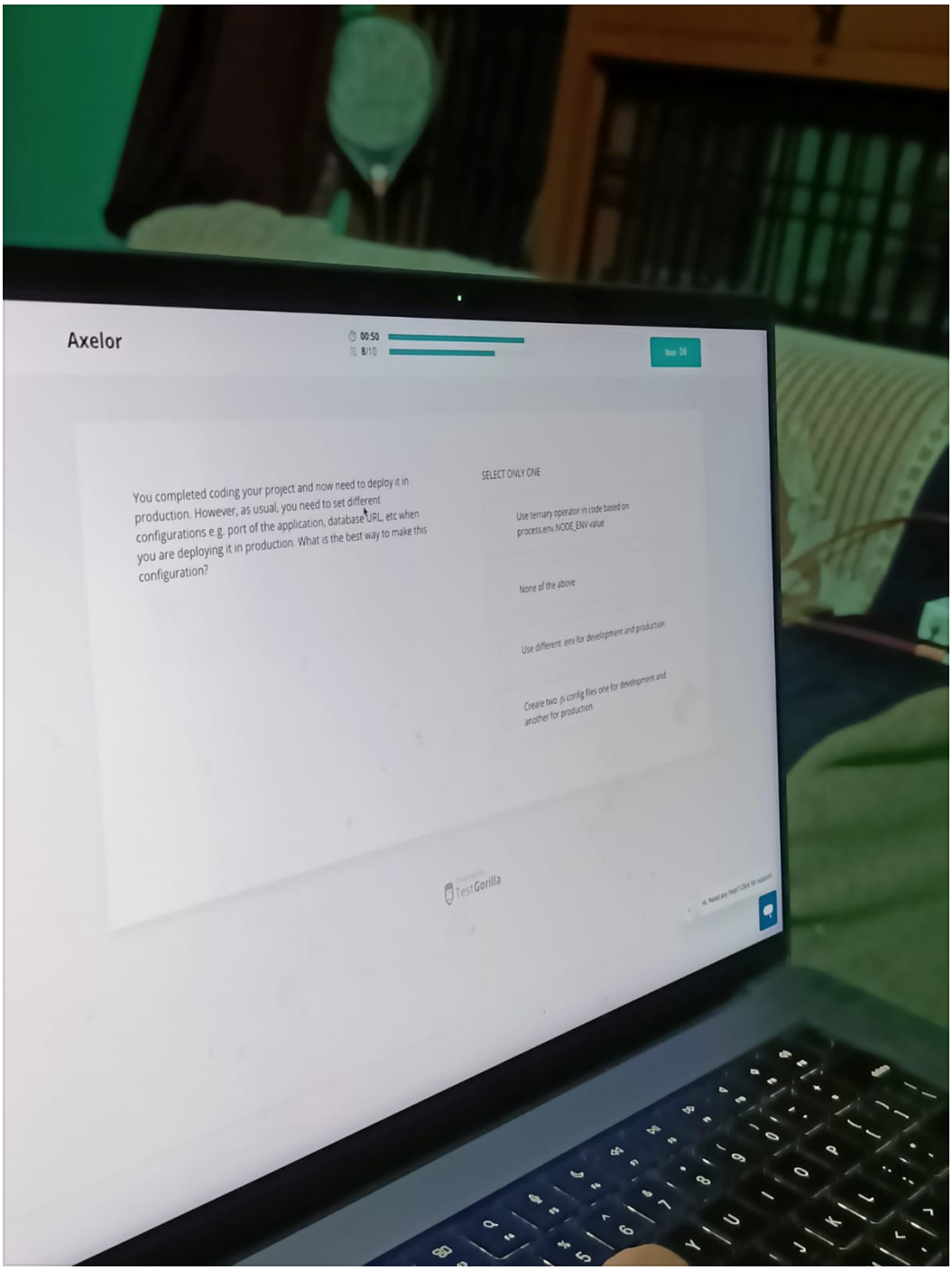
```
}).catch((error) => {
  console.error("Promise rejected:", error);
});

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

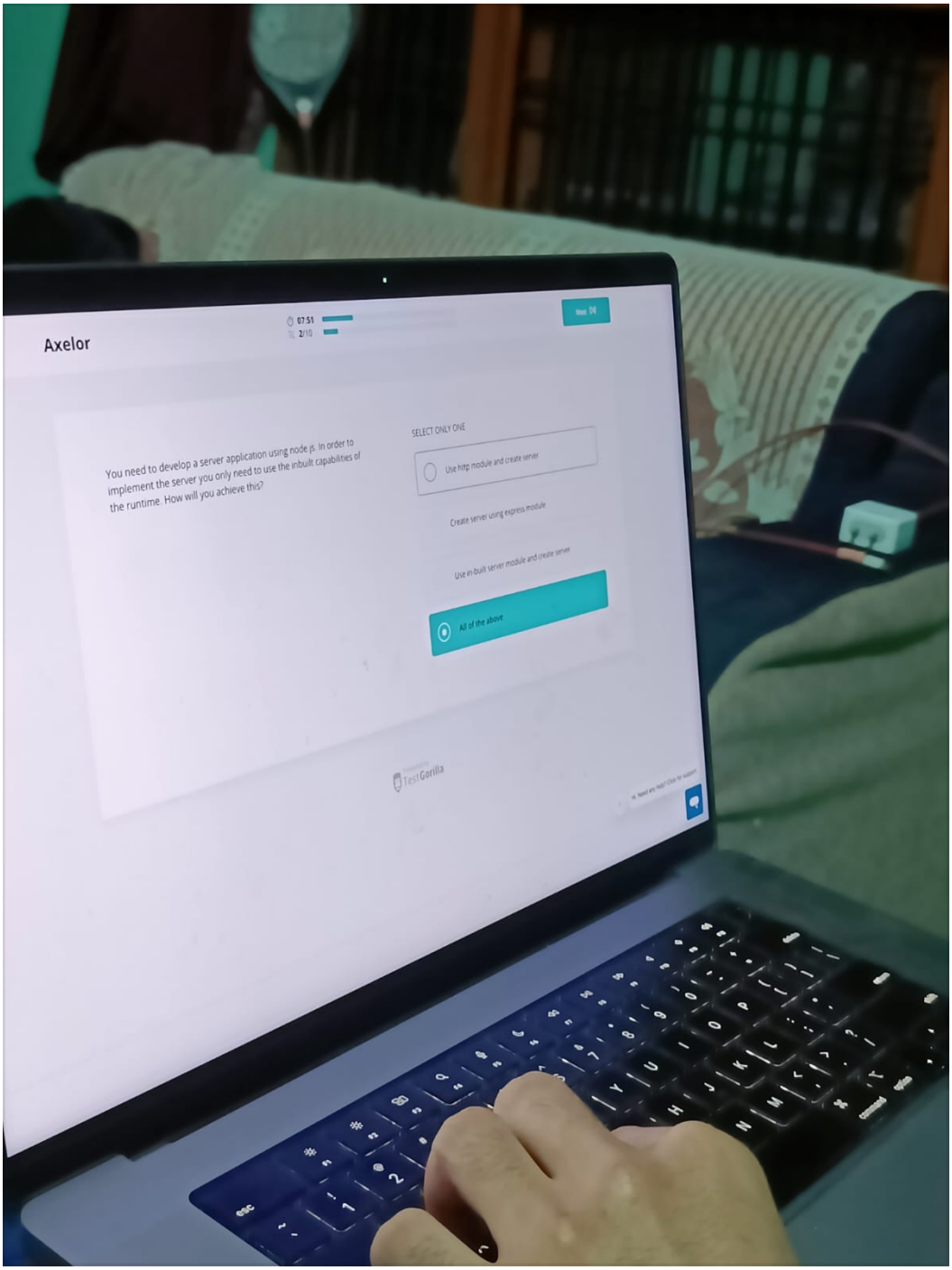


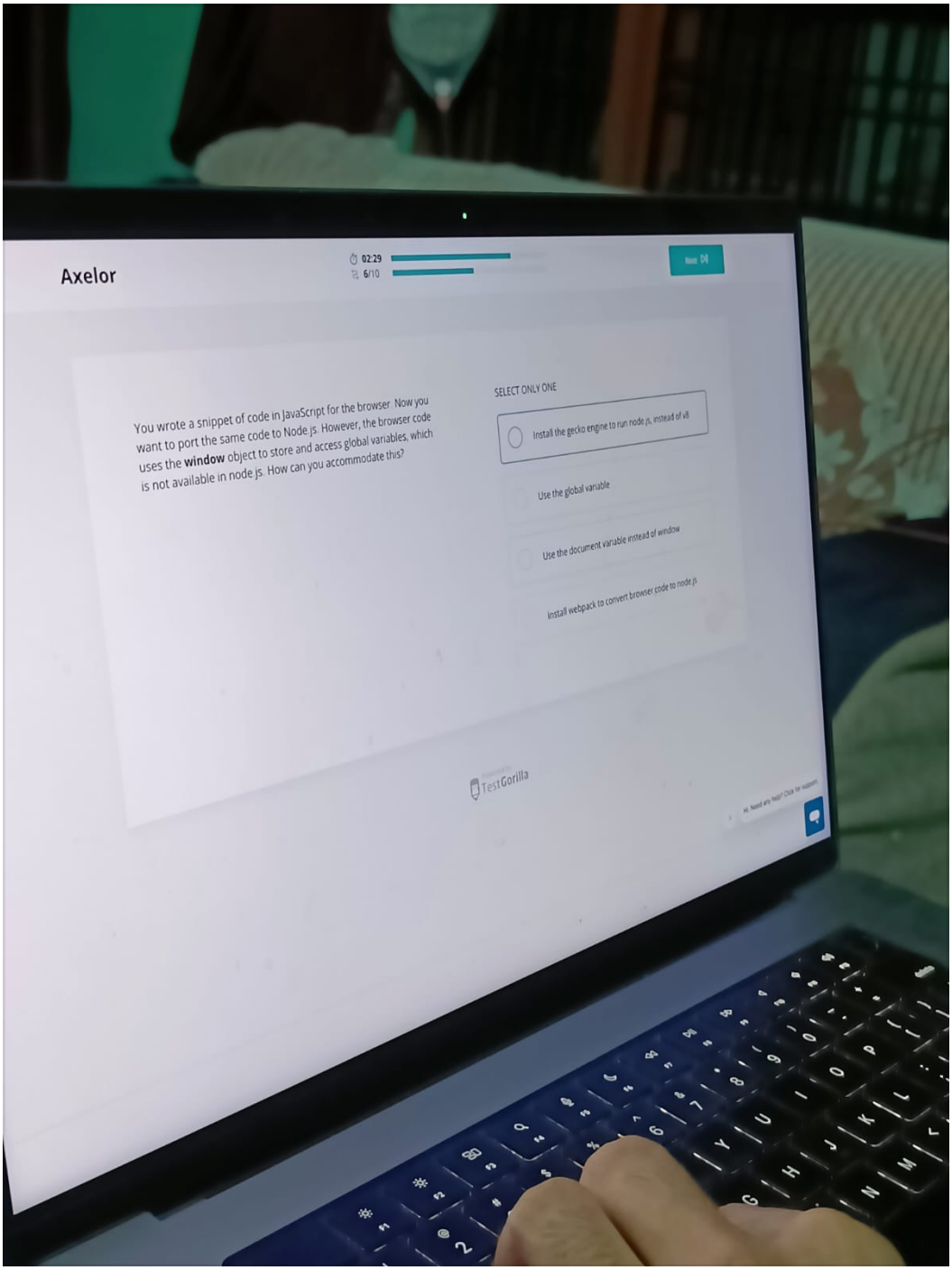

07:38
3/10
What will be the order of the console log for the code mentioned below
SELECT ONLY ONE
function foo() {
console.log("a");
setTimeout(() => {
console.log("b");
}, 0);
Promise.resolve().then(() => {
console.log("c");
});
}
foo();
a->c->b
Throws error because setTimeout is called with 0 ms
a->b->c
b->c->a

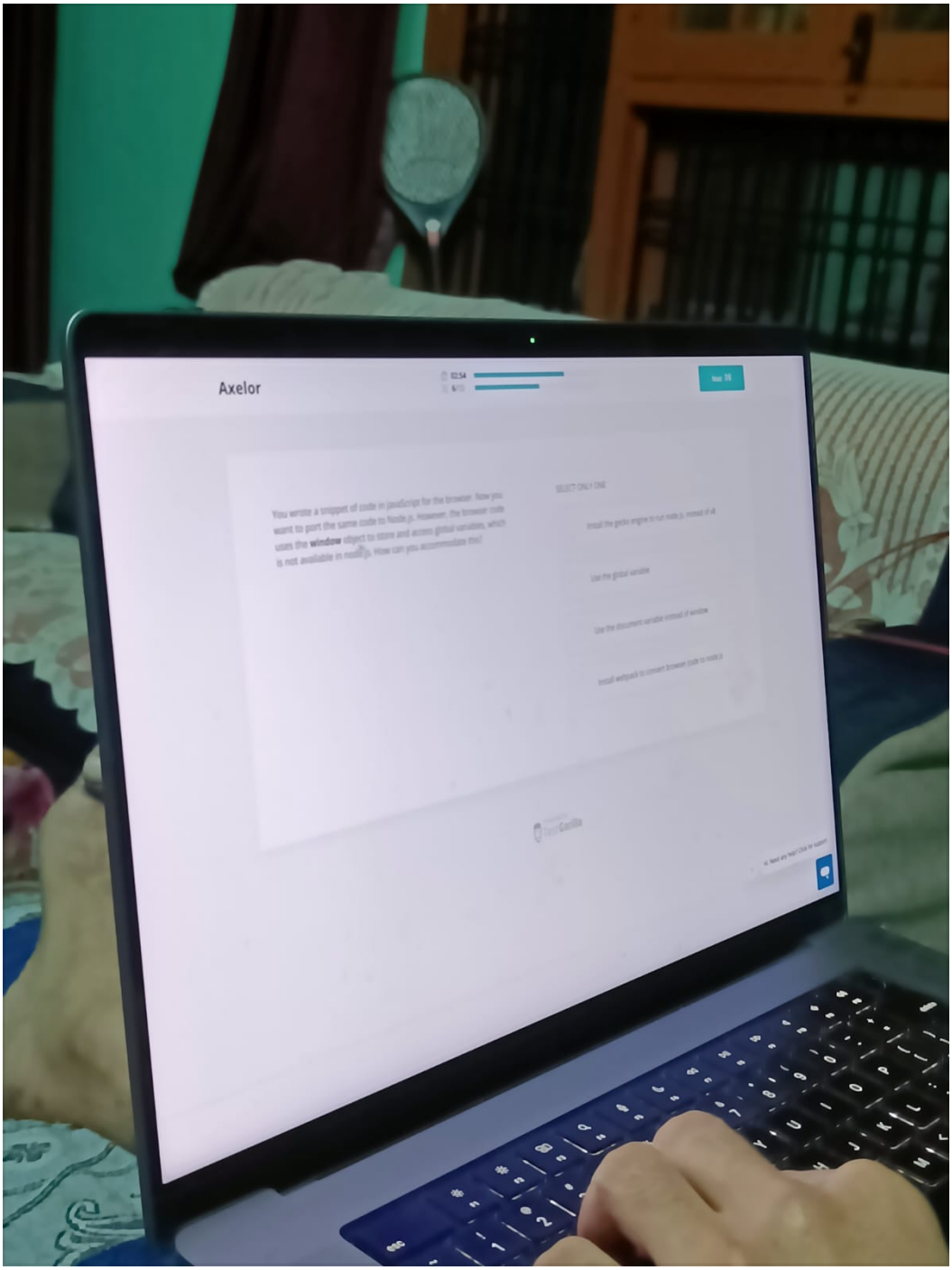

Axelor
04:50
4/10
SELECT ONLY ONE
You are making use of the .env file for setting environment variables for a project. You need to set custom variables in the file and later use them in your source code. How will you achieve getting custom variables from the .env file?
Install dotenv module and use config method from the same
You cannot set custom variables in your .env file
Create a custom function that uses fs module to read env content
You can import using require statement e.g const variable = require('.env')
TestGorilla

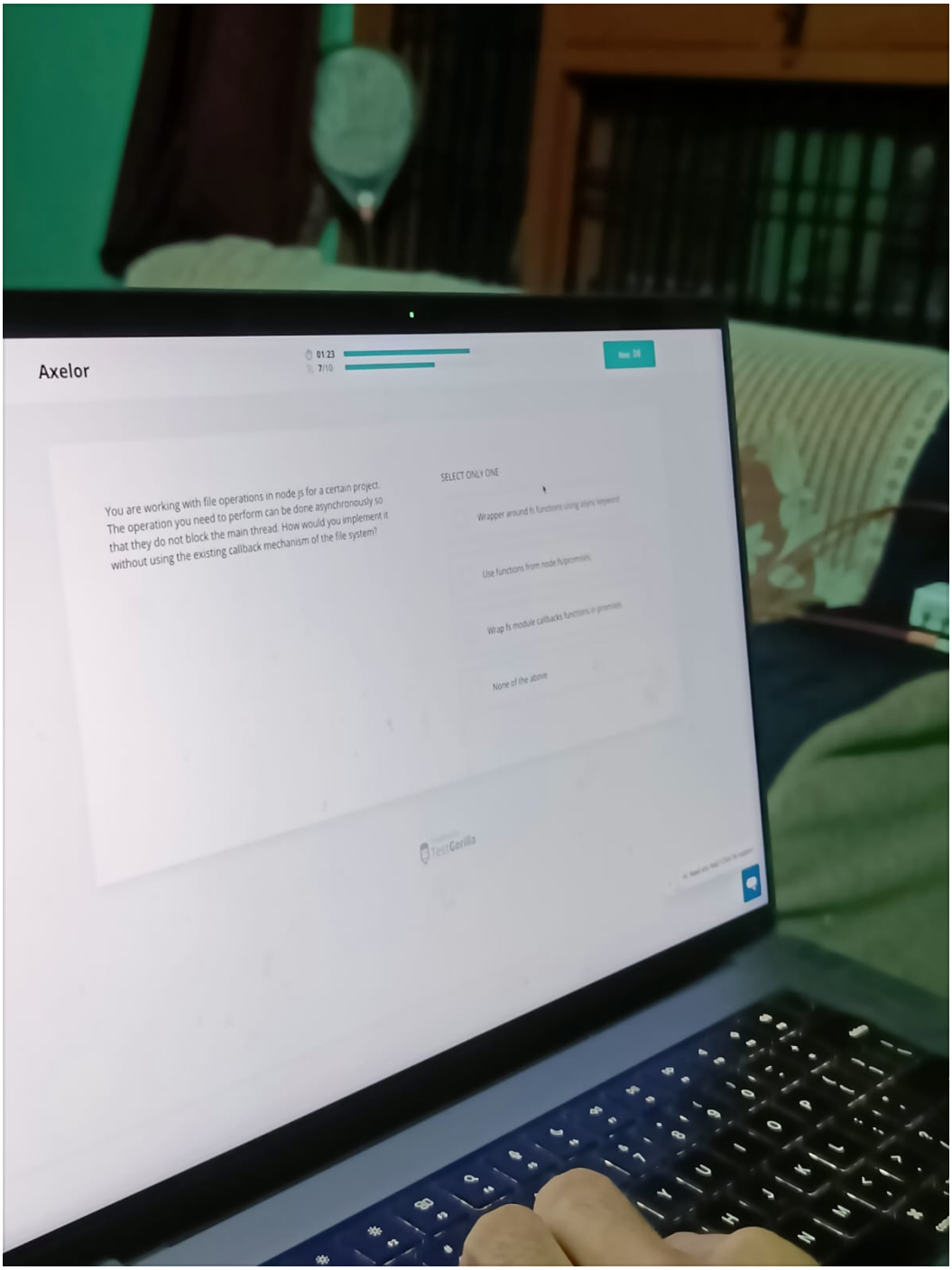

Axelor
09:58
1/10
You are working on creating a library project. You need to create multiple modules for the same. While importing the internal modules you generally use the commonjs format of importing using the require keyword. Instead you need to use the es6 module system and import the modules using import keyword.
SELECT ONLY ONE
Change the file extension to .mjs
Installing babel to support the es6 module
You can use es6 module without making any changes to the project
Changing type variable to module in package.json
TestGorilla

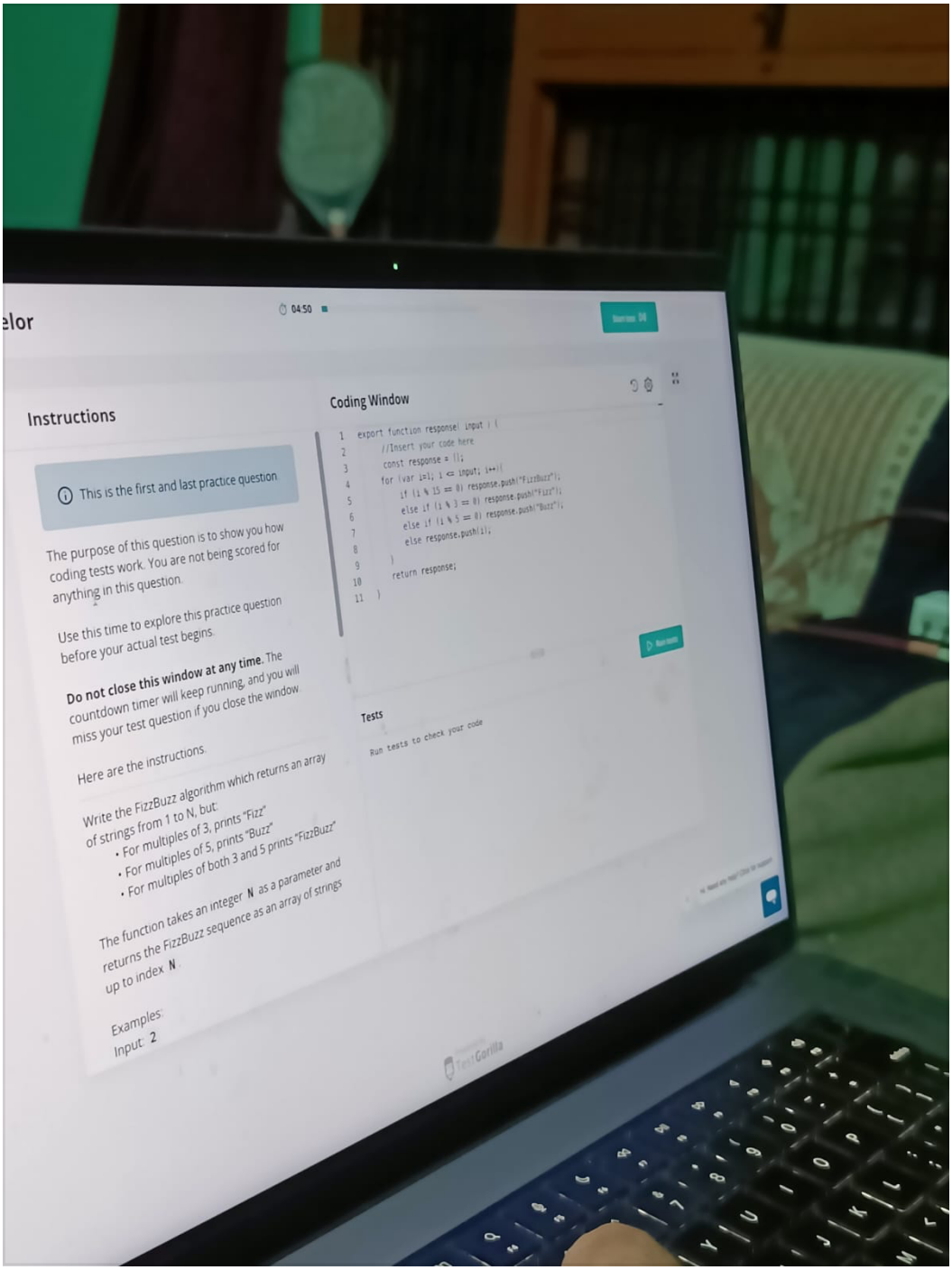

Axelor
00:50
8/10
You completed coding your project and now need to deploy it in production. However, as usual, you need to set different configurations e.g. port of the application, database URL, etc when you are deploying it in production. What is the best way to make this configuration?
SELECT ONLY ONE
Use ternary operator in code based on process.env.NODE_ENV value
None of the above
Use different .env for development and production
Create two .js config files one for development and another for production
TestGorilla

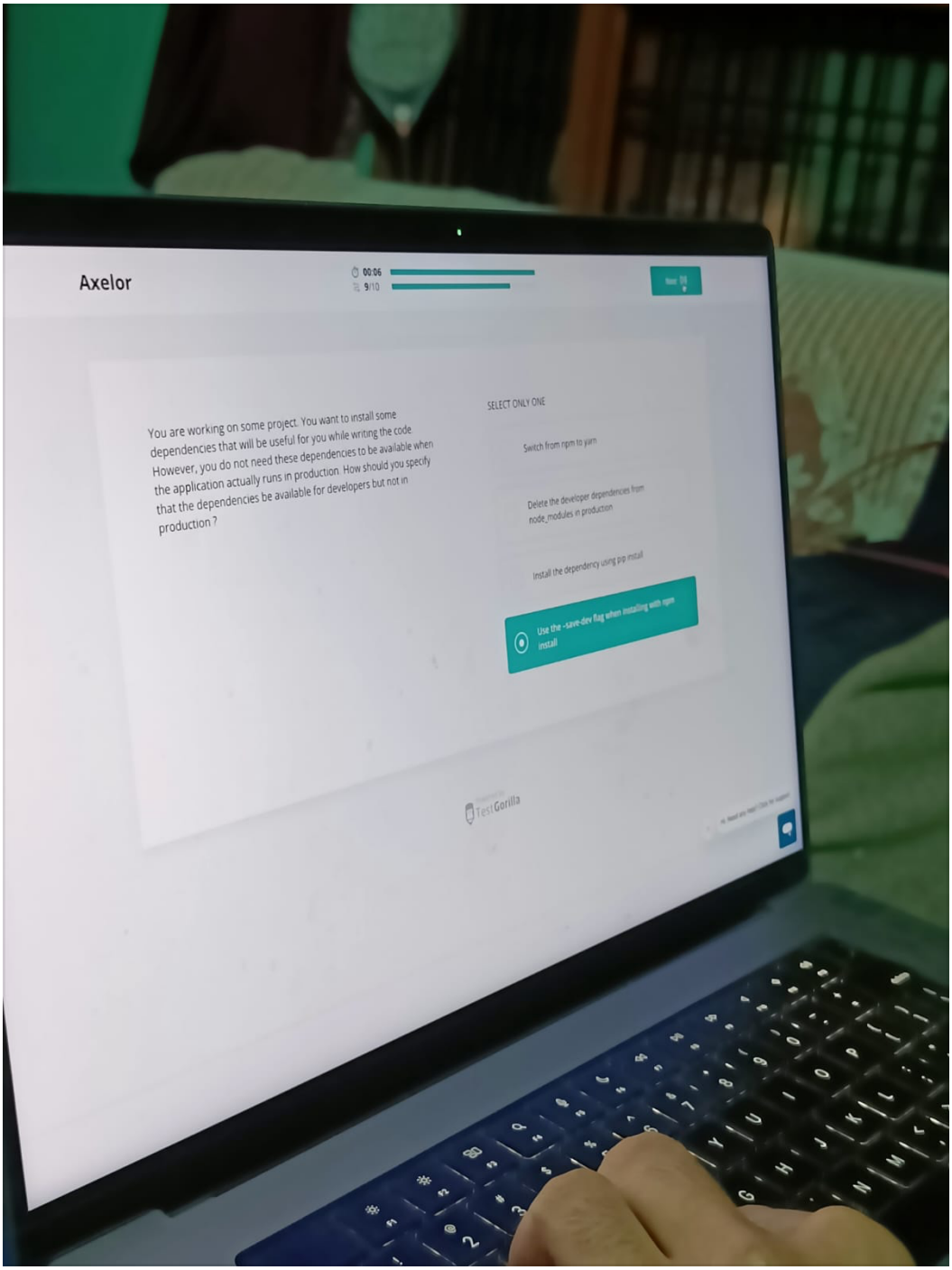

Axelor
07:51
2/10
You need to develop a server application using node.js. In order to implement the server you only need to use the inbuilt capabilities of the runtime. How will you achieve this?
SELECT ONLY ONE
Use http module and create server
Create server using express module
Use in-built server module and create server
All of the above
TestGorilla

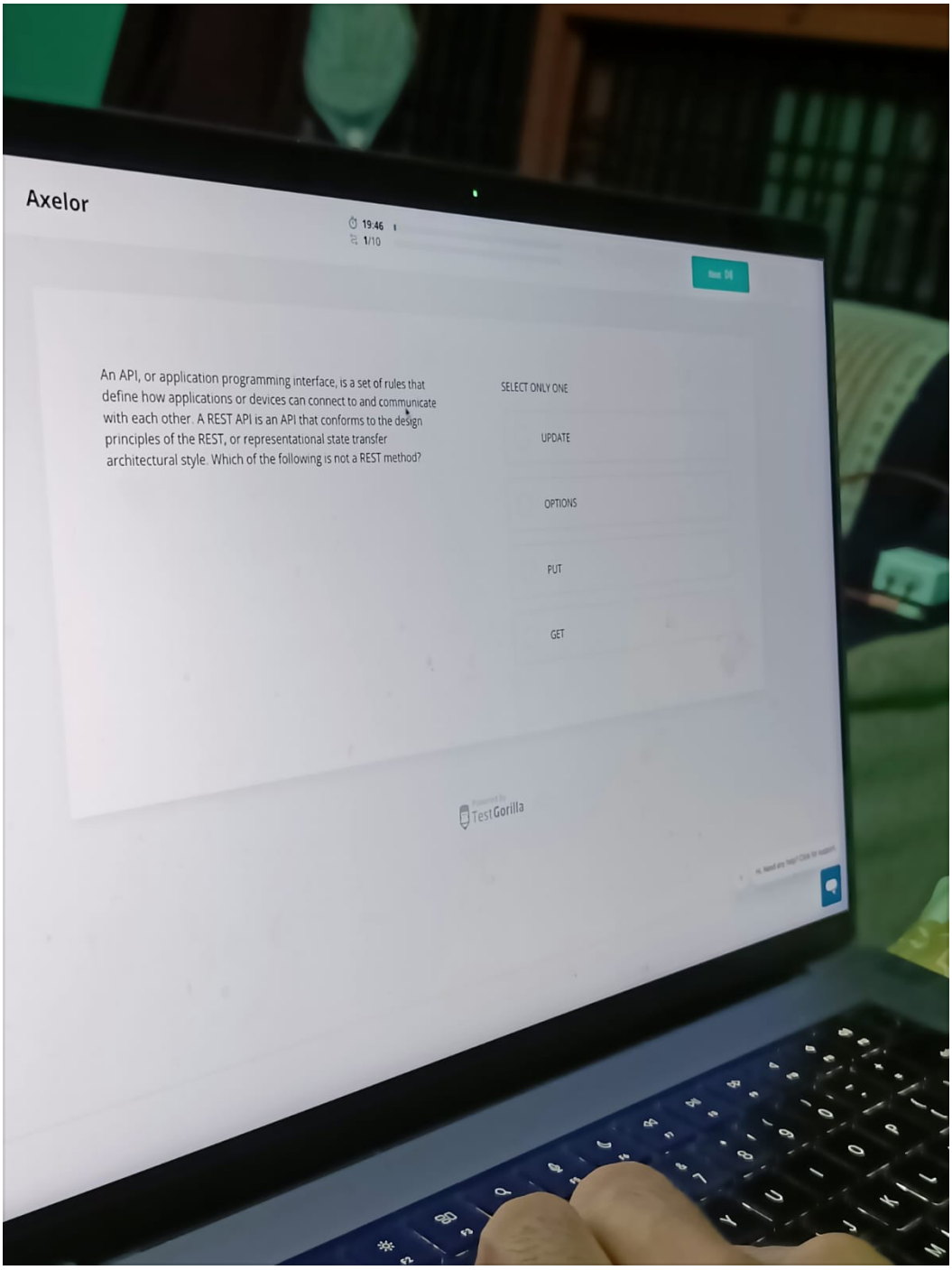

Axelor
02:29
6/10
SELECT ONLY ONE
You wrote a snippet of code in JavaScript for the browser. Now you want to port the same code to Node.js. However, the browser code uses the **window** object to store and access global variables, which is not available in node.js. How can you accommodate this?
Install the gecko engine to run node.js, instead of v8
Use the global variable
Use the document variable instead of window
Install webpack to convert browser code to node.js
TestGorilla

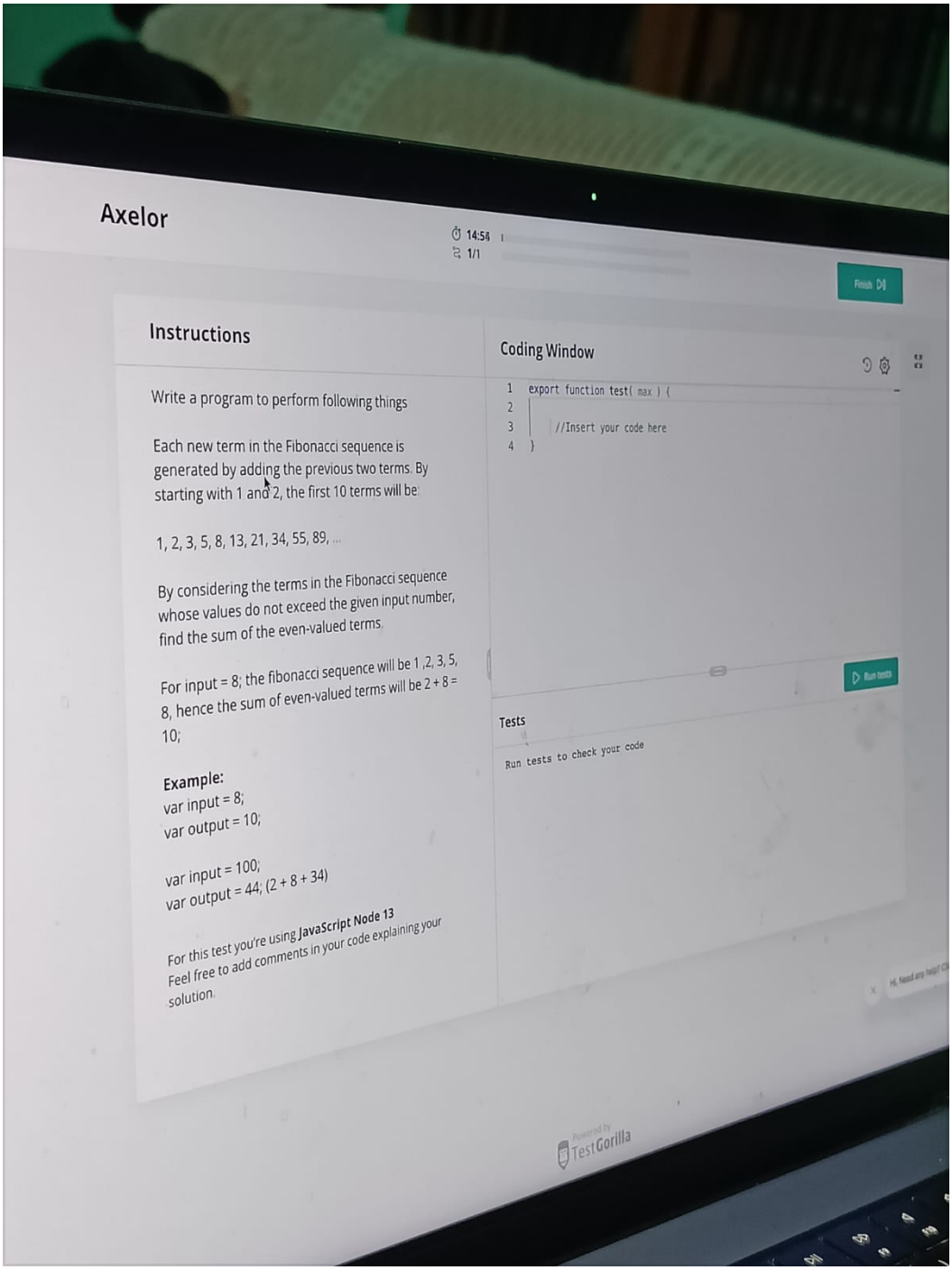

Axelor
You wrote a snippet of code in JavaScript for the browser. Now you want to port the same code to Node.js. However, the browser code uses the window object to store and access global variables, which is not available in Node.js. How can you accommodate this?
SELECT ONLY ONE
Use the global variable
Use the document variable instead of window

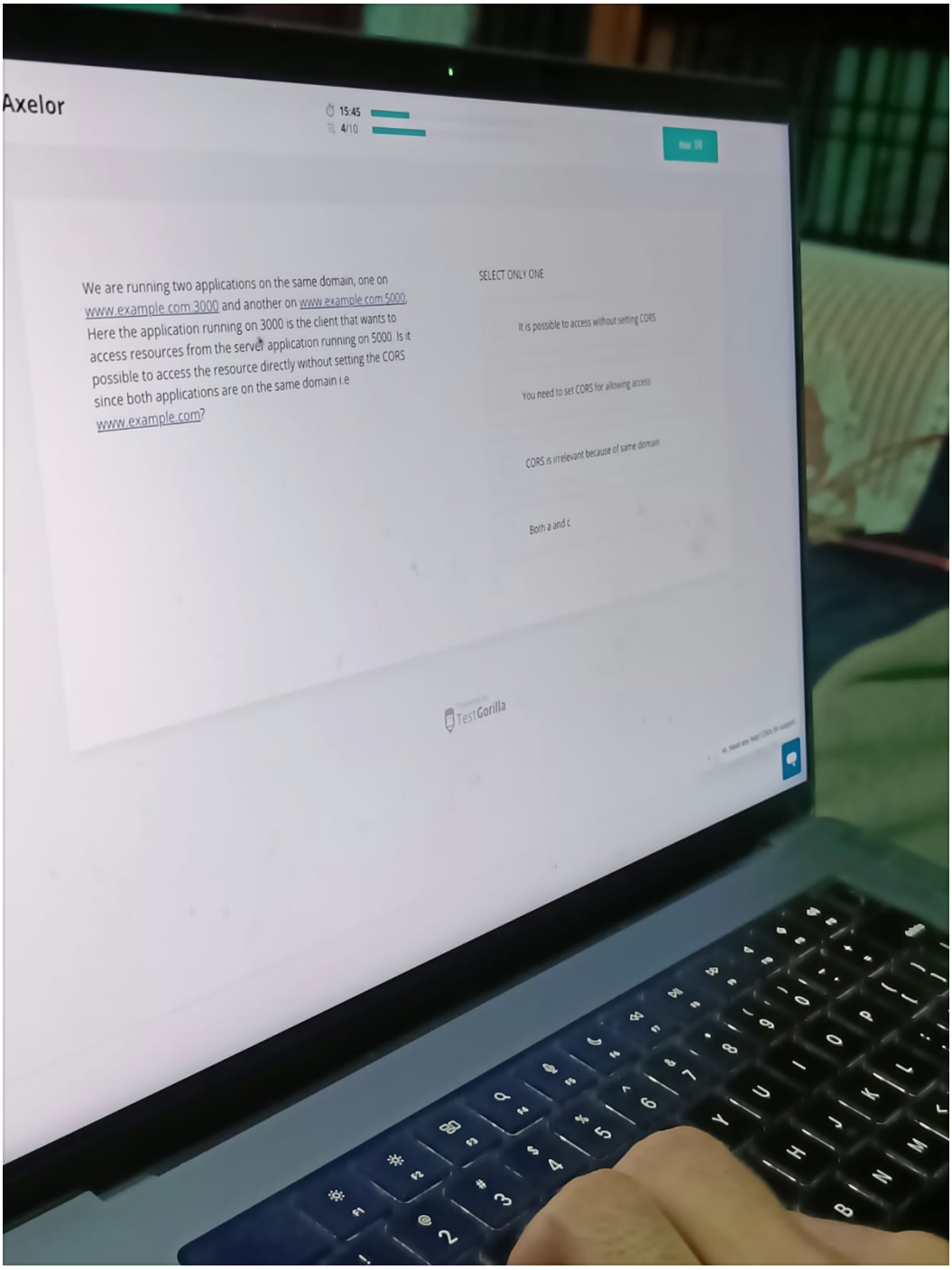

Axelor
01:23
7/10
You are working with file operations in node.js for a certain project. The operation you need to perform can be done asynchronously so that they do not block the main thread. How would you implement it without using the existing callback mechanism of the file system?
SELECT ONLY ONE
Use functions from node:fs/promises
Wrap fs module callbacks functions in promises
None of the above
TestGorilla

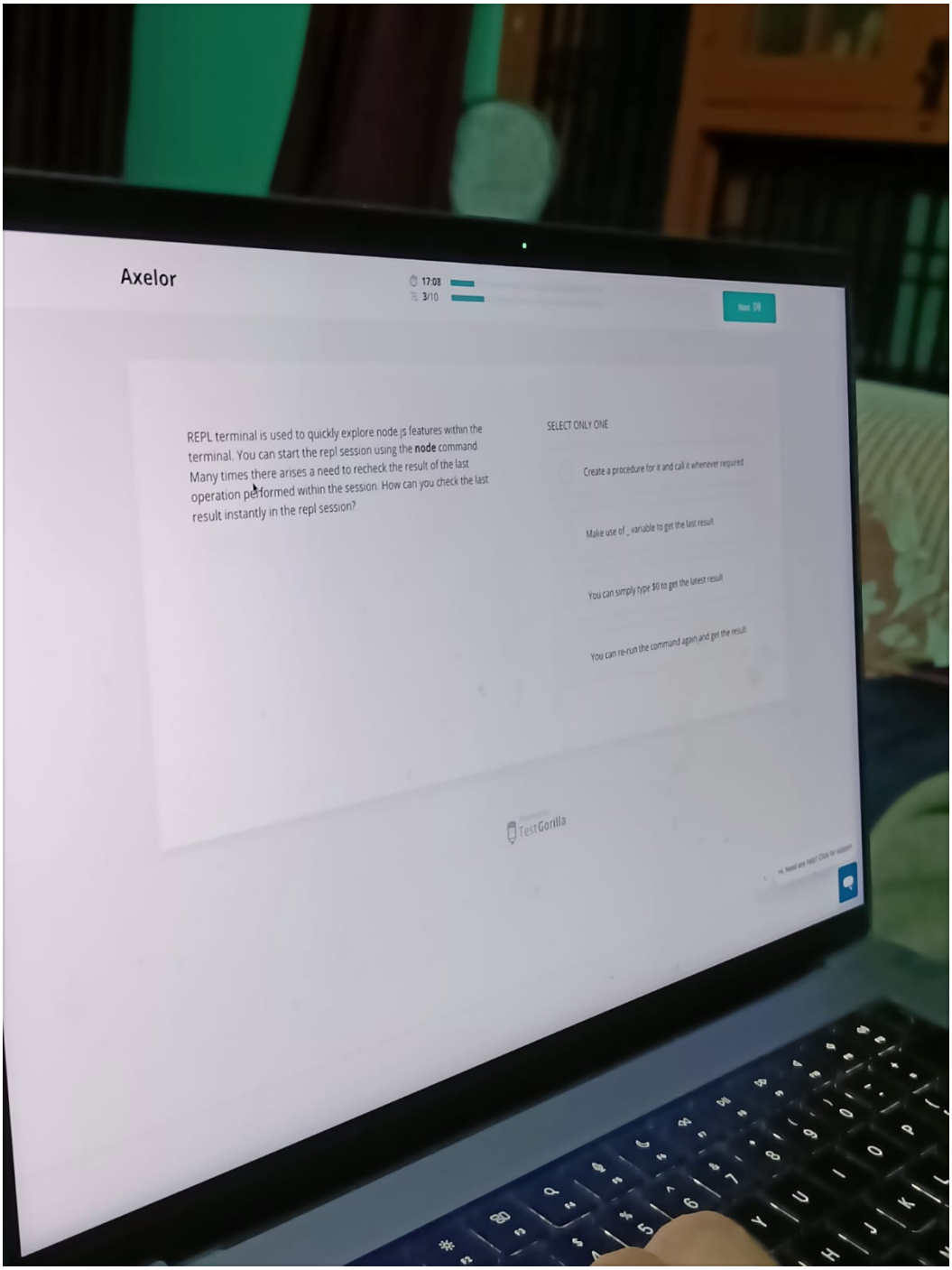

elor
04:50
Instructions
This is the first and last practice question.
The purpose of this question is to show you how coding tests work. You are not being scored for anything in this question.
Use this time to explore this practice question before your actual test begins.
Do not close this window at any time. The countdown timer will keep running, and you will miss your test question if you close the window.
Here are the instructions.
Write the FizzBuzz algorithm which returns an array of strings from 1 to N, but:
• For multiples of 3, prints "Fizz"
• For multiples of 5, prints "Buzz"
• For multiples of both 3 and 5 prints "FizzBuzz"
The function takes an integer N as a parameter and returns the FizzBuzz sequence as an array of strings up to index N
Examples:
Input: 2
Coding Window
export function response(input) {
//Insert your code here
const response = [];
for (var i=1; i <= input; i++){
if (i % 15 == 0) response.push("FizzBuzz");
else if (i % 3 == 0) response.push("Fizz");
else if (i % 5 == 0) response.push("Buzz");
else response.push(i);
}
return response;
}
Tests
Run tests to check your code
TestGorilla

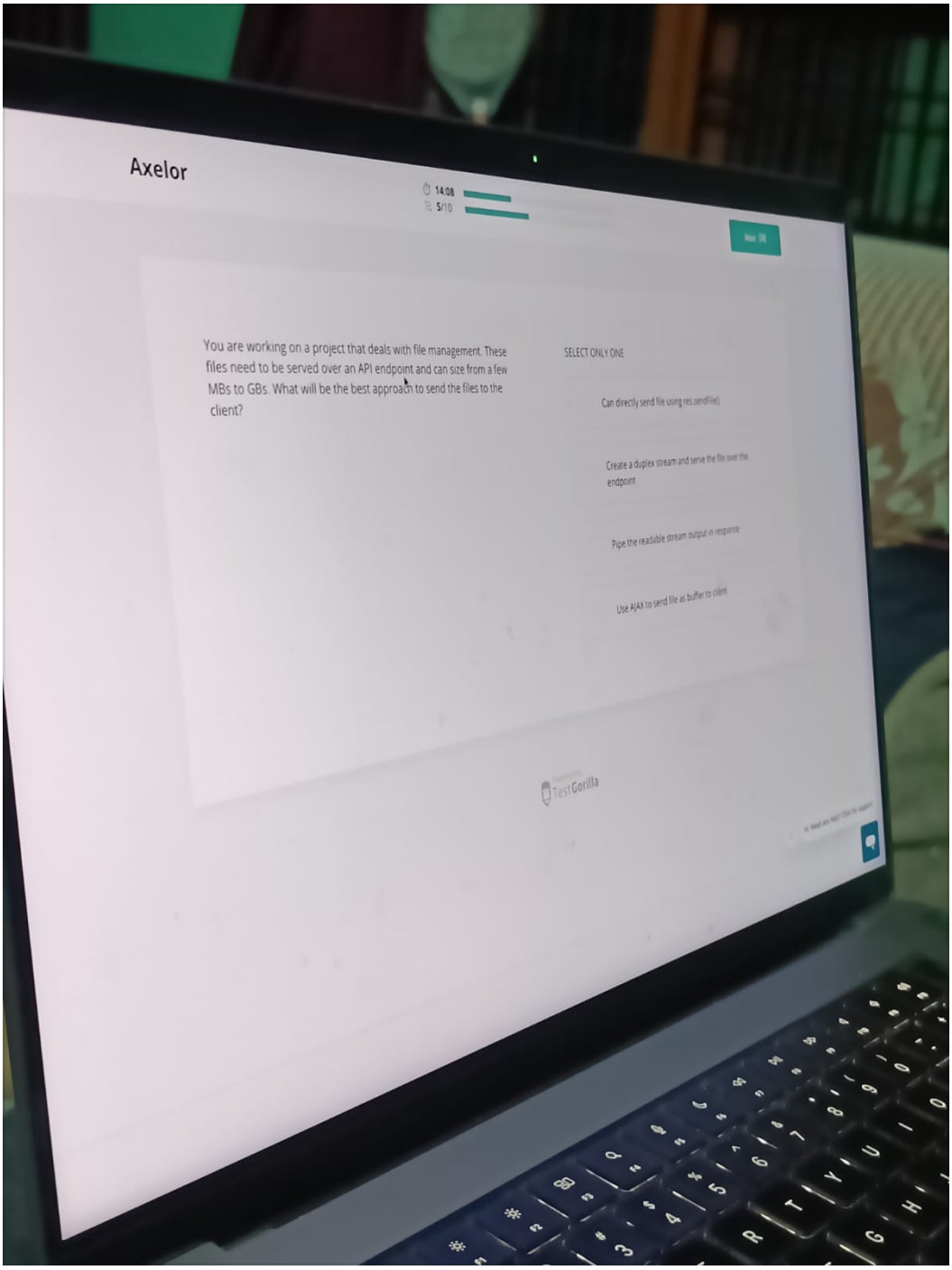

Axelor
00:06
9/10
You are working on some project. You want to install some dependencies that will be useful for you while writing the code. However, you do not need these dependencies to be available when the application actually runs in production. How should you specify that the dependencies be available for developers but not in production?
SELECT ONLY ONE
Switch from npm to yarn
Delete the developer dependencies from node_modules in production
Install the dependency using pip install
Use the --save-dev flag when installing with npm install
TestGorilla

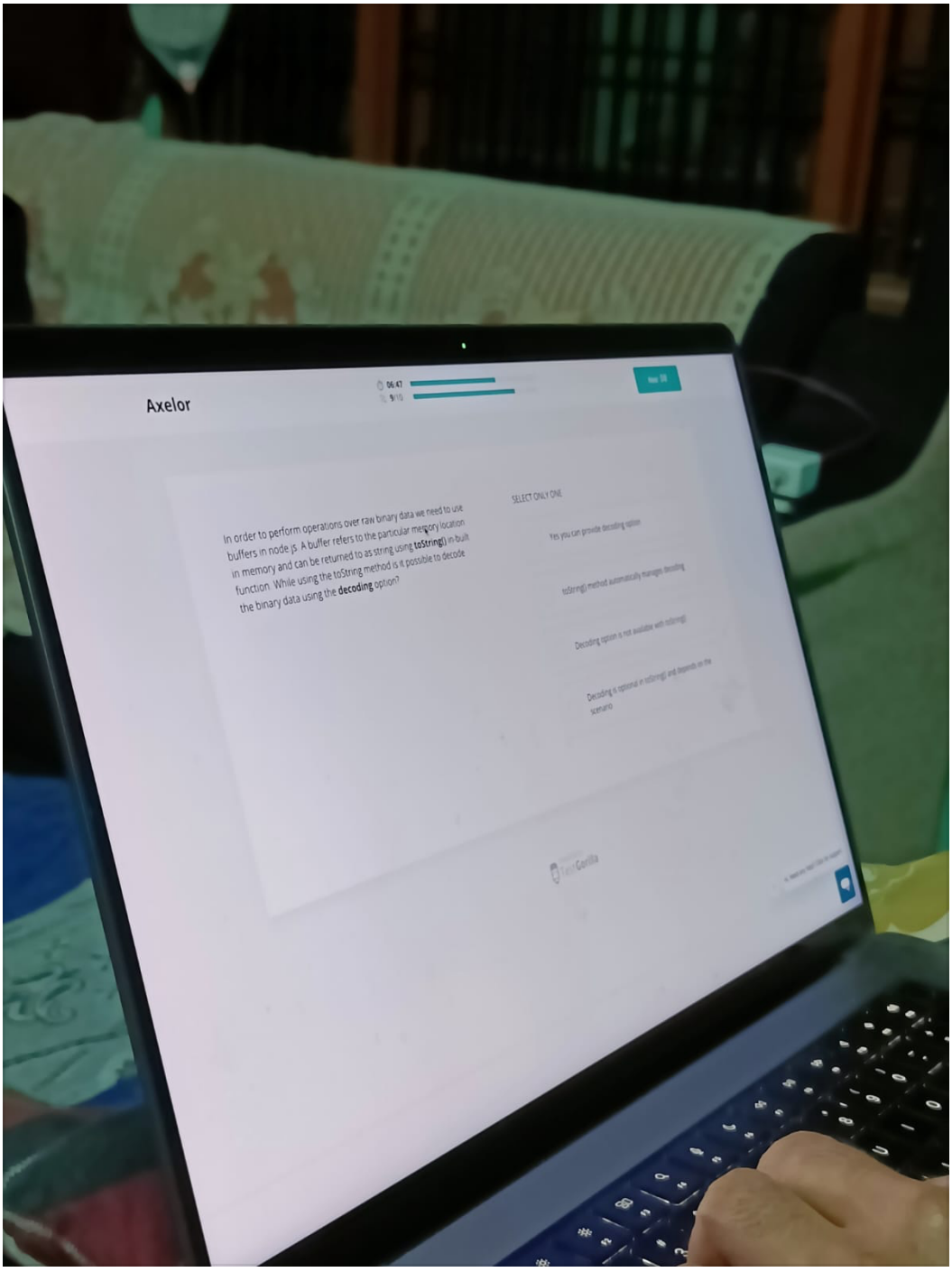

Axelor
19:46
1/10
An API, or application programming interface, is a set of rules that define how applications or devices can connect to and communicate with each other. A REST API is an API that conforms to the design principles of the REST, or representational state transfer architectural style. Which of the following is not a REST method?
SELECT ONLY ONE
UPDATE
OPTIONS
PUT
GET
TestGorilla

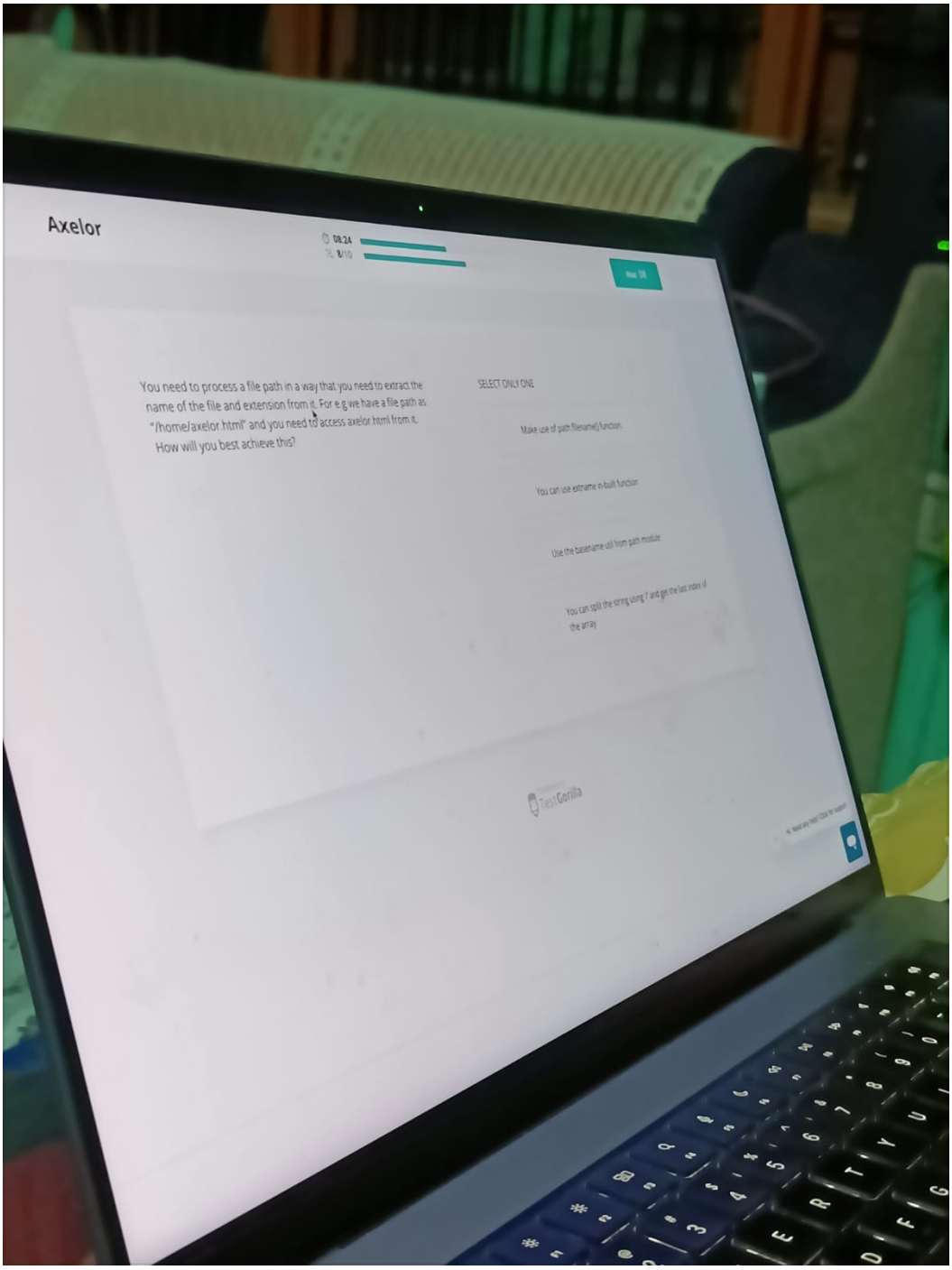

Axelor
14:58
1/1
Finish
Instructions
Write a program to perform following things
Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:
1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
By considering the terms in the Fibonacci sequence whose values do not exceed the given input number, find the sum of the even-valued terms.
For input = 8; the fibonacci sequence will be 1 ,2, 3, 5, 8, hence the sum of even-valued terms will be 2 + 8 = 10;
Example:
var input = 8;
var output = 10;
var input = 100;
var output = 44; (2 + 8 + 34)
For this test you're using JavaScript Node 13
Feel free to add comments in your code explaining your solution.
Coding Window
1 export function test(max) {
2
3 //Insert your code here
4 }
Run tests
Tests
Run tests to check your code
Powered by
TestGorilla

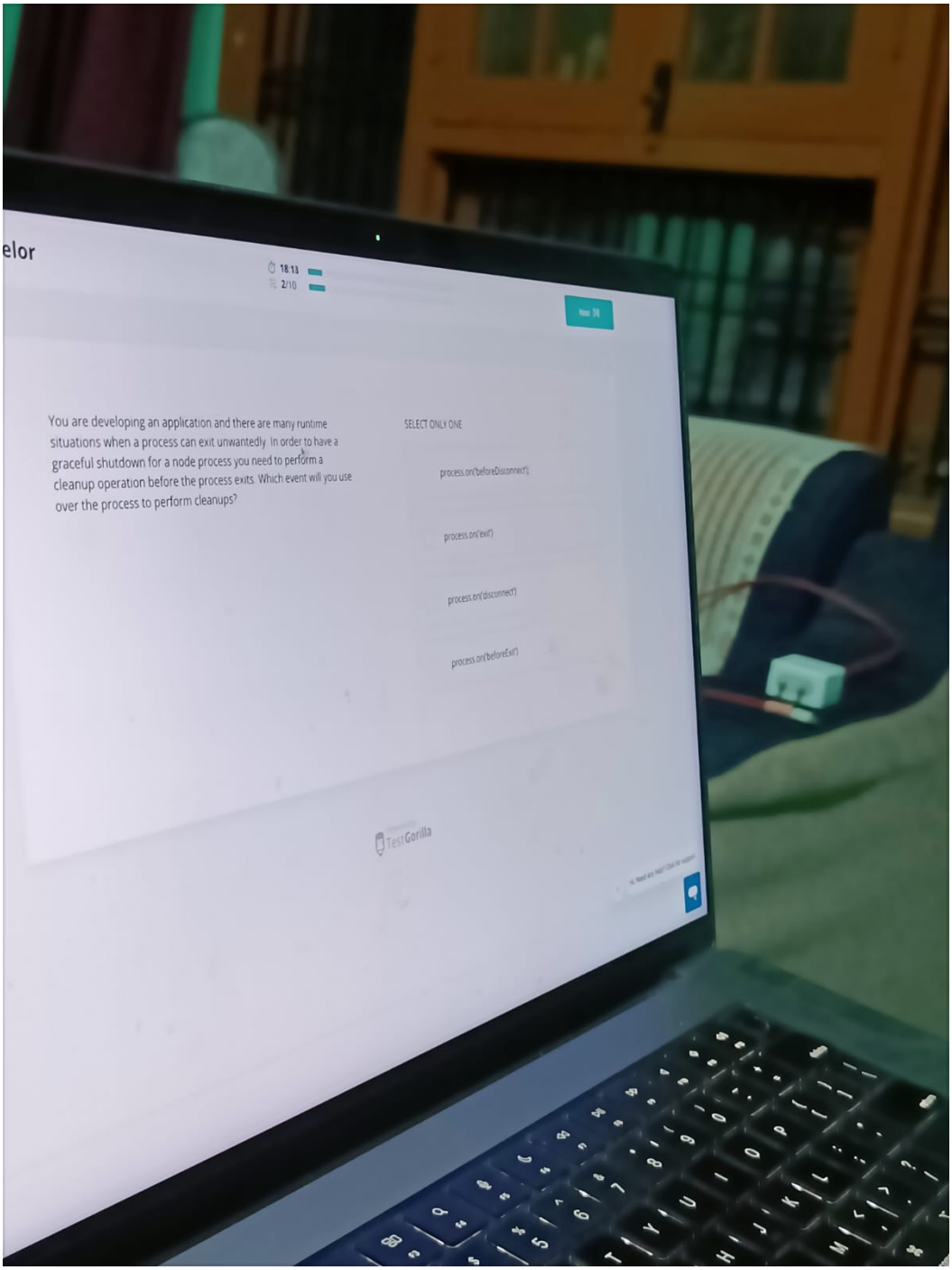

Axelor
15:45
4/10
We are running two applications on the same domain, one on www.example.com:3000 and another on www.example.com:5000. Here the application running on 3000 is the client that wants to access resources from the server application running on 5000. Is it possible to access the resource directly without setting the CORS since both applications are on the same domain i.e www.example.com?
SELECT ONLY ONE
It is possible to access without setting CORS
You need to set CORS for allowing access
CORS is irrelevant because of same domain
Both a and c
TestGorilla

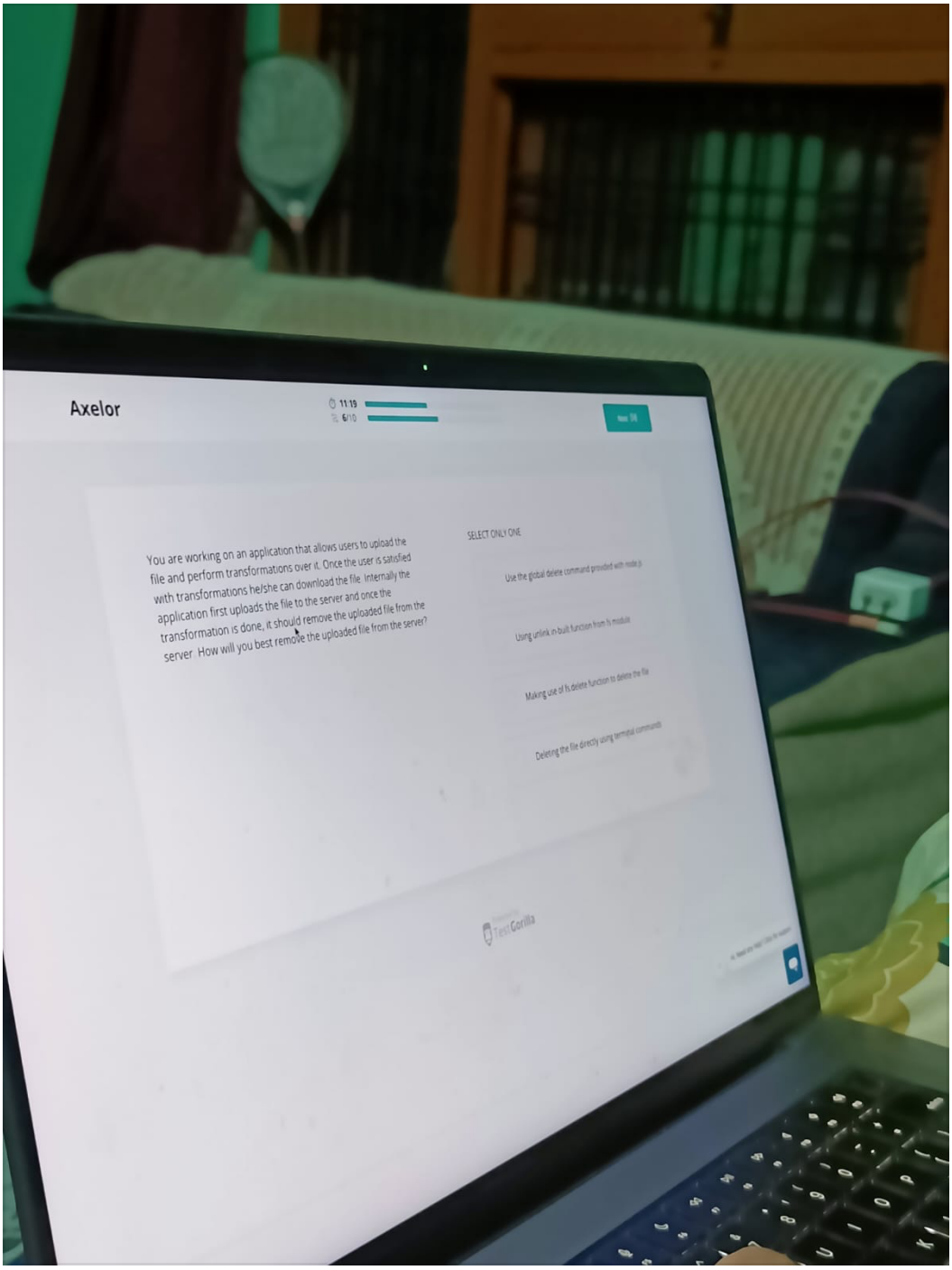

Axelor
17:08
3/10
REPL terminal is used to quickly explore node.js features within the terminal. You can start the repl session using the node command. Many times there arises a need to recheck the result of the last operation performed within the session. How can you check the last result instantly in the repl session?
SELECT ONLY ONE
Create a procedure for it and call it whenever required
Make use of _ variable to get the last result
You can simply type $0 to get the latest result
You can re-run the command again and get the result
TestGorilla

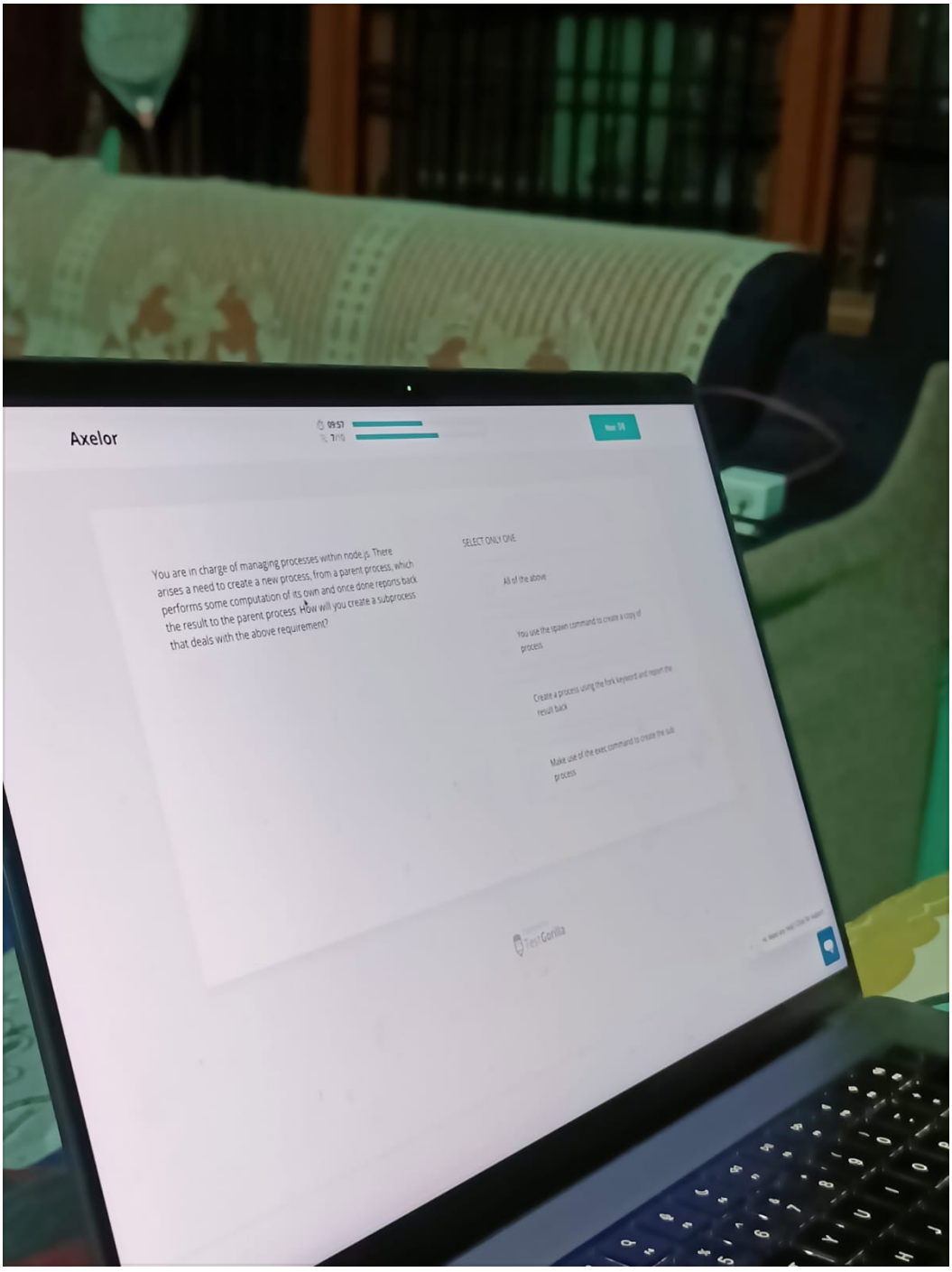

Axelor
14:08
5/10
You are working on a project that deals with file management. These files need to be served over an API endpoint and can size from a few MBs to GBs. What will be the best approach to send the files to the client?
SELECT ONLY ONE
Can directly send file using res.sendFile()
Create a duplex stream and serve the file over the endpoint
Pipe the readable stream output in response
Use AJAX to send file as buffer to client
TestGorilla


Axelor
06:47
In order to perform operations over raw binary data we need to use buffers in node.js. A buffer refers to the particular memory location in memory and can be returned to as string using toString() in-built function. While using the toString method is it possible to decode the binary data using the decoding option?
SELECT ONLY ONE
Yes you can provide decoding option
toString() method automatically manages decoding
Decoding option is not available with toString()
Decoding is optional in toString() and depends on the scenario


Axelor
08:24
You need to process a file path in a way that you need to extract the name of the file and extension from it. For e.g we have a file path as "/home/axelor.html" and you need to access axelor.html from it. How will you best achieve this?
SELECT ONLY ONE
Make use of path.filename() function.
You can use extname in-built function
Use the basename util from path module
You can split the string using '/' and get the last index of the array
TestGorilla


elor
18:13
2/10
You are developing an application and there are many runtime situations when a process can exit unwantedly. In order to have a graceful shutdown for a node process you need to perform a cleanup operation before the process exits. Which event will you use over the process to perform cleanups?
SELECT ONLY ONE
process.on('beforeDisconnect');
process.on('exit')
process.on('disconnect')
process.on('beforeExit')
TestGorilla


Axelor
11:19
6/10
You are working on an application that allows users to upload the file and perform transformations over it. Once the user is satisfied with transformations he/she can download the file. Internally the application first uploads the file to the server and once the transformation is done, it should remove the uploaded file from the server. How will you best remove the uploaded file from the server?
SELECT ONLY ONE
Use the global delete command provided with node.js
Using unlink, in-built function from fs module
Making use of fs.delete function to delete the file
Deleting the file directly using terminal commands
TestGorilla


Axelor
09:57
7/10
Next
You are in charge of managing processes within node.js. There arises a need to create a new process, from a parent process, which performs some computation of its own and once done reports back the result to the parent process. How will you create a subprocess that deals with the above requirement?
SELECT ONLY ONE
All of the above
You use the spawn command to create a copy of process
Create a process using the fork keyword and report the result back
Make use of the exec command to create the sub process
TestGorilla