1. Basics of Express.js

1. What is Express.js? Why use it?

Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for building web and mobile applications, as well as APIs. It's built on top of Node.js's built-in `http` module.

We use it because it simplifies the process of building web applications by providing a layer of abstraction over raw Node.js. Instead of manually handling low-level details like routing, parsing requests, and handling headers, Express.js provides a clean, easy-to-use API to do all that. This saves a lot of development time and effort. It's also **unopinionated**, meaning it doesn't force you into a specific architecture or set of libraries, allowing you to choose the tools that best fit your project.

2. Difference between Node.js and Express.js.

Aspect	**Node.js**	**Express.js**
Category	Runtime Environment	Web Framework
Purpose	To run JavaScript on the server	To simplify web application development on Node.js
Core	Based on the V8 engine	Built on top of Node.js's `http` module
Features	I/O operations, event-driven architecture, file system access	Routing, middleware, templating, request/response handling
Use Case	Building command-line tools, server-side applications, and network apps from scratch	Building web apps and APIs with pre-built features and structure

Think of Node.js as the **engine** of a car, and Express.js as the **chassis** and **body**. You can build a car with just the engine, but Express.js provides the structure, steering wheel, and pedals that make it much easier to drive and more functional.

3. What are the key features of Express.js?

- **Robust routing:** Easily define different routes for handling HTTP requests.
- **Middleware support:** A powerful and extensible system for handling requests and responses.
- **Templating engines:** Support for popular templating engines like Pug, EJS, and Handlebars.
- **Static file serving:** Built-in functionality to serve static assets like CSS, JavaScript, and images.
- **HTTP utility methods:** A wide range of methods for handling requests and sending responses.
- **High performance:** Express.js is fast and efficient, making it suitable for high-traffic applications.

4. How to install and create a basic Express.js application?

1. **Install Node.js:** Make sure you have Node.js installed on your system.
2. **Initialize a new project:** Navigate to your project directory and run `npm init -y` to create a `package.json` file.
3. **Install Express.js:** Run `npm install express`.
4. **Create a server file** (e.g., `app.js`):

```
JavaScript

// app.js
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello, World!');
});

app.listen(port, () => {
  console.log(`Express app listening at http://localhost:${port}`);
});
```

5. **Run the server:** Run the file using `node app.js`. Your server will be running on `http://localhost:3000`.

5. What is middleware in Express.js? Types of middleware?

Middleware is a function that has access to the **request object (`req`)**, the **response object (`res`)**, and the **next middleware function** in the application's request-response cycle. It can:

- Execute any code.
- Make changes to the request and response objects.
- End the request-response cycle.
- Call the next middleware function in the stack.

Types of Middleware:

- **Application-level middleware:** Binds to an instance of the `app` object using `app.use()`. It runs for every request unless a specific path is defined.
- **Router-level middleware:** Functions the same as application-level middleware but is bound to an instance of `express.Router()`.
- **Error-handling middleware:** Always takes four arguments: `(err, req, res, next)`. It is specifically for handling errors.
- **Built-in middleware:** Comes with Express.js, like `express.static`, `express.json`, and `express.urlencoded`.
- **Third-party middleware:** Middleware functions loaded using `npm`, like `morgan` (for logging) or `body-parser` (for parsing request bodies).

6. What is the difference between application-level and router-level middleware?

- **Application-level middleware** is loaded using `app.use()`. It's global to the application or a specific path, and it affects all routes defined after it.
- JavaScript

```
const app = express();
app.use((req, res, next) => {
  console.log('This middleware runs for all requests.');
  next();
});
```

-
-
- **Router-level middleware** is loaded using `router.use()`. It is scoped to a specific `express.Router()` instance, allowing you to create modular, reusable middleware for a group of routes.
- JavaScript

```
const router = express.Router();
router.use((req, res, next) => {
  console.log('This middleware only runs for routes in this router.');
  next();
});
```

-
-

7. How to serve static files in Express.js?

You use the built-in `express.static` middleware. You pass the directory name where your static files (images, CSS, JS) are located to the middleware.

Example:

```
JavaScript

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

// Serve files from the 'public' directory
app.use(express.static('public'));

// Your 'public' directory would contain files like:
// public/
// ├── images/
// └── style.css
```

If you then navigate to http://localhost:3000/style.css, Express will serve the file.

8. What are routes in Express.js? How to define them?

Routes determine how an application responds to a client request to a specific endpoint (a URI) and an HTTP request method (GET, POST, etc.).

You define routes using methods like app.get(), app.post(), app.put(), app.delete(), etc. These methods take a path and a callback function (the **route handler**).

Example:

```
JavaScript

// GET request to the root path
app.get('/', (req, res) => {
  res.send('Welcome to the home page.');
});

// POST request to a /users path
app.post('/users', (req, res) => {
  res.send('User created.');
});
```

9. Difference between app.get(), app.post(), app.put(), app.delete().

These are all **HTTP request methods** and are used to define route handlers for specific actions, following the **RESTful API** convention.

- app.get(): Used to **retrieve** data from a server. It should be idempotent (multiple identical requests have the same effect as a single one).
- app.post(): Used to **send** data to a server to create a new resource.
- app.put(): Used to **update** a resource. It's also idempotent, as it replaces the entire resource.
- app.delete(): Used to **delete** a resource from the server.

10. What is `app.listen()` in Express.js?

`app.listen()` is a method that starts a server and makes it listen for connections on the specified port and host. It's the final step to make your Express application accessible. It's a thin wrapper around Node.js's native `http.listen()` method.

Example:

```
JavaScript

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

---

2. Middleware & Routing

11. What is the role of `next()` in middleware?

The `next()` function is a crucial part of the middleware pattern. When called, it passes control to the **next middleware function** in the application's request-response cycle. If a middleware function does not end the cycle (e.g., by calling `res.send()`), it **must** call `next()` to avoid leaving the request hanging.

Example:

```
JavaScript

app.use((req, res, next) => {
  console.log('Middleware 1 is running...');
  req.requestTime = Date.now(); // Adding a property to the request object
  next(); // Pass control to the next function
});

app.get('/home', (req, res) => {
  res.send(`Home page requested at: ${req.requestTime}`);
});
```

Without `next()`, the first middleware would block the request and the `/home` route handler would never be reached.

12. How does error-handling middleware work in Express.js?

Error-handling middleware is a special type of middleware that takes **four arguments**: `(err, req, res, next)`. Express recognizes a function as an error handler if it has this signature. When an error occurs (either explicitly passed to `next(err)` or an unhandled promise rejection), Express bypasses the regular middleware stack and jumps directly to the first error-handling middleware.

Example:

```
JavaScript

app.get('/error', (req, res, next) => {
  // Simulate an error
  next(new Error('Something went wrong!'));
});

// Error-handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack); // Log the error stack for debugging
  res.status(500).send('Something broke!');
});
```

13. How to handle 404 errors in Express.js?

To handle 404 errors (resource not found), you should define a middleware function at the very end of your middleware stack, **after all other routes**. This ensures it's only reached if no other route has handled the request.

Example:

```
JavaScript

// ... all your routes go here ...

// This middleware will be executed if no other route matched
app.use((req, res, next) => {
  res.status(404).send("Sorry, can't find that!");
});
```

14. What is `app.use()` vs `app.all()`?

- `app.use()`: Used to **mount middleware** functions at a specific path. It works for **all HTTP methods** (GET, POST, PUT, etc.) and is often used for things like static file serving, logging, or authentication.
- `app.all()`: Used to define a route handler that responds to **all HTTP methods** (GET, POST, PUT, DELETE, etc.) at a specific path. It's useful when you need to perform the same logic for multiple HTTP verbs on the same endpoint.

Example:

```
JavaScript

// app.use() - Middleware for all methods on /admin
app.use('/admin', (req, res, next) => {
  // Authentication logic here
```

```
  next();
});

// app.all() - Route handler for all methods on /secret
app.all('/secret', (req, res) => {
  res.send('Accessing the secret route with any method.');
});
```

15. Difference between `req.params`, `req.query`, and `req.body`.

These are all properties of the `request` object used to access different parts of an HTTP request.

- `req.params`: Used to access **route parameters**, which are part of the URL path. They are defined in the route with a colon (`:`).
    - **Example URL:** `/users/123`
    - **Route:** `app.get('/users/:id', ...)`
    - **Access:** `req.params.id` would be `'123'`.
- `req.query`: Used to access **query string parameters**, which are appended to the URL after a question mark (`?`).
    - **Example URL:** `/search?q=nodejs&sort=asc`
    - **Route:** `app.get('/search', ...)`
    - **Access:** `req.query.q` would be `'nodejs'` and `req.query.sort` would be `'asc'`.
- `req.body`: Used to access data sent in the **request body**, typically from a POST or PUT request (e.g., form data, JSON). It requires middleware like `express.json()` or `express.urlencoded()` to parse the body.
    - **Example:** A POST request with `{ "name": "John" }` in the body.
    - **Access:** `req.body.name` would be `'John'`.

16. How to create modular routes using `express.Router()`?

`express.Router()` is a powerful feature for creating **modular, mountable route handlers**. It allows you to organize your routes into separate files, making your application more maintainable and scalable.

Example:

Create a file users.js for user-related routes:

JavaScript

```
// users.js
const express = require('express');
const router = express.Router();

// Define user routes on the router
router.get('/', (req, res) => {
  res.send('List of all users.');
```

```
});

router.post('/', (req, res) => {
  res.send('New user created.');
});

module.exports = router;
```

Then, in your main `app.js` file, you can mount this router:

```
JavaScript

// app.js
const express = require('express');
const app = express();
const usersRouter = require('./users'); // Import the router

// Mount the user router at the /users path
app.use('/users', usersRouter);
```

Now, a GET request to `/users` will be handled by the router you defined in `users.js`.

17. What are chained route handlers in Express.js?

Chained route handlers are a way to define multiple handlers for the **same route path** but for **different HTTP methods** using a single statement. This is done using the `app.route()` method. It helps in avoiding duplicate route path strings and can be more efficient.

Example:

```
JavaScript

app.route('/book')
  .get((req, res) => {
    res.send('Get a random book');
  })
  .post((req, res) => {
    res.send('Add a book');
  })
  .put((req, res) => {
    res.send('Update the book');
  });
```

18. Difference between synchronous and asynchronous middleware.

- **Synchronous Middleware:** A middleware function that performs a blocking operation. It **does not** take a `next()` argument and simply returns after its task is complete, or it calls the next middleware function directly.

    - **Example:** A simple logger that doesn't involve I/O.
- **Asynchronous Middleware:** A middleware function that performs a non-blocking operation, typically involving I/O (like a database query or an API call). The function must call `next()` when the asynchronous operation is complete to pass control to the next function.
    - **Example:**
- JavaScript

```
app.use(async (req, res, next) => {
  try {
    await someAsyncFunction();
    next();
  } catch (err) {
    next(err);
  }
});
```

-
- **Note:** Express has built-in support for `async/await` in middleware, where if you `await` an operation, the `next()` call is implicitly handled once the promise resolves.

19. How to use third-party middleware (like `body-parser`, `morgan`)?

1. **Install the middleware:** Use `npm` to install the package.
    - `npm install morgan`
    - `npm install body-parser` (Note: `body-parser` is now a part of Express.js itself, so `express.json()` and `express.urlencoded()` are the modern way).
2. **`require` the package** in your `app.js` file.
3. **`app.use()` the middleware** to add it to your application's middleware stack.

Example with `morgan` for logging:

JavaScript

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

// Use morgan with the 'dev' format
app.use(morgan('dev'));

app.get('/', (req, res) => {
  res.send('Hello!');
});
```

20. How to implement global vs local middleware?

- **Global middleware** is applied to the **entire application**. You implement it by calling `app.use()` without a specific path, or with a path that's a prefix for all your routes (e.g., `app.use('/')`).
    - **Example:** Logging or authentication for all requests.
- JavaScript

```
app.use(express.json()); // Global middleware for parsing JSON
app.use(morgan('tiny')); // Global middleware for logging
```

-
-
- **Local middleware** is applied only to a **specific route or a group of routes**. You implement it by passing the middleware function as an argument to the route handler.
    - **Example:** A specific validation middleware for a user creation route.
- JavaScript

```
const validateUser = (req, res, next) => {
  // ... validation logic ...
  next();
};

app.post('/users', validateUser, (req, res) => {
  // This handler will only run if validateUser calls next()
  res.send('User is valid!');
});
```

-
-

---

3. Request & Response Handling

21. How to handle form data in Express.js?

To handle form data, you need to use middleware to parse the request body. The most common types are URL-encoded data from HTML forms and JSON data from modern APIs.

- **For URL-encoded data** (from a standard HTML form), use `express.urlencoded()`:
- JavaScript

```
app.use(express.urlencoded({ extended: true }));
// Now, you can access form data like this:
app.post('/profile', (req, res) => {
  console.log(req.body.username);
});
```

-
-
- **For JSON data** (from a JavaScript frontend or API client), use `express.json()`:

- JavaScript

```
app.use(express.json());
// Now, you can access JSON data like this:
app.post('/api/data', (req, res) => {
  console.log(req.body.someData);
});
```

-
-

22. Difference between `res.send()`, `res.json()`, and `res.end()`.

- `res.send()`: A high-level method that sends a response of various types. It automatically sets the `Content-Type` header based on the argument. It can send a string, a buffer, an object, or an array. It also handles setting the ETag and Content-Length headers.
- `res.json()`: Specifically for sending a JSON response. It converts the object or array to a JSON string and sets the `Content-Type` header to `application/json`. It's a convenience method and the preferred way to send JSON data.
- `res.end()`: A low-level method that terminates the response process. It does not send any data and should be avoided in most cases in favor of `res.send()` or `res.json()`. It's useful when you want to end a request without sending any content, for example, in a middleware function.

23. What is `res.redirect()` in Express.js?

`res.redirect()` is a method that redirects a user's browser to a different URL. It takes the path as an argument. You can also specify a status code (e.g., `301` for permanent, `302` for temporary).

Example:

JavaScript

```
app.get('/old-route', (req, res) => {
  res.redirect('/new-route'); // Redirects to '/new-route'
});
```

24. What is `res.render()`? How to use templating engines (EJS, Pug, Handlebars)?

`res.render()` is a method used to render a view template. It takes the name of the template file and an optional object of data to pass to the template. Express needs a **templating engine** to know how to process these templates.

How to use:

1. **Install a templating engine:** `npm install ejs`

2. **Configure Express:** Tell Express which view engine to use and where the view files are located.
3. JavaScript

```
const express = require('express');
const app = express();
app.set('view engine', 'ejs');
app.set('views', './views'); // Set the directory for view files
```

4.
5.
6. **Create a view file** (e.g., views/index.ejs):
7. HTML

```
<h1>Hello, <%= name %>!</h1>
```

8.
9.
10. **Render the view** in a route handler:
11. JavaScript

```
app.get('/', (req, res) => {
  res.render('index', { name: 'World' });
});
```

12.
13.

This will send an HTML response with "Hello, World!" to the client.

25. How to handle file uploads in Express.js? (Multer middleware).

Handling file uploads directly in Express is complex because it involves multipart/form-data. The **Multer middleware** simplifies this process.

1. **Install Multer:** npm install multer
2. **Require and configure Multer:**
3. JavaScript

```
const multer = require('multer');
const upload = multer({ dest: 'uploads/' }); // `dest` is the destination folder
```

4.
5.
6. **Use Multer in your route:** Multer provides middleware to handle different types of uploads.
    - upload.single('file'): For a single file.
    - upload.array('files'): For multiple files.

- upload.fields(...): For different fields.

7. JavaScript

```
app.post('/upload', upload.single('myFile'), (req, res, next) => {
  // req.file contains the uploaded file details
  console.log(req.file);
  res.send('File uploaded!');
});
```

8.
9. The uploaded file will be saved in the uploads/ directory, and req.file will contain its metadata.

26. How to handle cookies in Express.js?

To handle cookies, you need the **cookie-parser middleware**. It parses cookies from the request headers and makes them available on the req.cookies object.

1. **Install cookie-parser:** npm install cookie-parser
2. **Use the middleware:**
3. JavaScript

```
const cookieParser = require('cookie-parser');
app.use(cookieParser());
```

4.
5.
6. **Set and read cookies:**
   - To **set a cookie**, use res.cookie():
   - JavaScript

```
res.cookie('name', 'value', { maxAge: 900000 });
```

   -
   -
   - To **read a cookie**, use req.cookies:
   - JavaScript

```
app.get('/', (req, res) => {
  console.log('Cookies:', req.cookies.name);
});
```

   -
   -

27. Difference between `req.cookies` and `req.signedCookies`.

Both are properties of the `request` object provided by `cookie-parser`.

- `req.cookies`: Contains the **unsigned** cookies sent by the client. These cookies are plain text and can be easily modified by the client.
- `req.signedCookies`: Contains **signed** cookies. These cookies are tamper-proof because a secret key is used to sign them. If a client tampers with the cookie, Express will detect the tampered signature and the cookie will not be available in `req.signedCookies`. To use signed cookies, you must provide a secret key to the middleware: `app.use(cookieParser('your-secret-key'));`.

28. How to set and clear cookies in Express.js?

- **Setting a cookie:** Use `res.cookie(name, value, [options])`.
- JavaScript

```
res.cookie('theme', 'dark', { httpOnly: true, secure: true });
```

- **Options:**
  - `httpOnly`: Prevents client-side JavaScript from accessing the cookie. This is a security best practice.
  - `secure`: Ensures the cookie is only sent over HTTPS.
  - `maxAge`: Sets the cookie expiration time in milliseconds.
- **Clearing a cookie:** Use `res.clearCookie(name, [options])`.
- JavaScript

```
res.clearCookie('theme');
```

- **Note:** The options object passed to `res.clearCookie()` should match the options used to set the cookie.

29. What is `express-session`? How to implement session management?

`express-session` is a middleware for creating and managing a user session. A session is a way to store data on the server that is specific to a single user. It works by assigning a unique **session ID** to each user, which is stored in a cookie. This ID is then used to look up the corresponding session data on the server.

How to implement:

1. **Install `express-session`:** `npm install express-session`
2. **Configure the middleware:**
3. JavaScript

```
const session = require('express-session');
```

```
app.use(session({
  secret: 'a-super-secret-key',
  resave: false,
  saveUninitialized: true,
  cookie: { secure: true } // Use `secure: true` in production with HTTPS
}));
```

4.
5.
6. **Store and access session data** in a route handler via `req.session`:
7. JavaScript

```
app.get('/', (req, res) => {
  req.session.views = (req.session.views || 0) + 1;
  res.send(`You have visited this page ${req.session.views} times.`);
});
```

8.
9.

30. How to parse JSON and URL-encoded data in Express.js?

Express has built-in middleware for this, making the `body-parser` library largely redundant for these two formats.

- **To parse JSON data:**
- JavaScript

```
app.use(express.json());
```

-
-
- **To parse URL-encoded data:**
- JavaScript

```
app.use(express.urlencoded({ extended: true }));
```

-
- The `extended: true` option uses the `qs` library, which allows for richer objects and arrays to be encoded in the URL.

---

4. Authentication & Security

31. How to implement authentication in Express.js?

Authentication in Express involves a few key steps:

1. **User Registration:** A route to handle new user sign-ups, typically saving a hashed password to the database.

2. **Login:** A route to verify user credentials (username/email and password). You should **never** store passwords in plain text; always use a strong hashing algorithm like `bcrypt`.
3. **Session/Token Management:** Once a user is authenticated, create a session or issue a token (like a JWT) to maintain their authenticated state across requests.
4. **Protected Routes:** Use middleware to check for the session or token on subsequent requests to protected routes.

32. Difference between session-based and token-based authentication.

- **Session-based Authentication:**
  - **Mechanism:** Stores user data on the **server** in a session store (e.g., in-memory, Redis, MongoDB). A session ID is sent to the client in a cookie.
  - **State: Stateful**. The server needs to maintain the session state for each logged-in user.
  - **Scalability:** Can be harder to scale horizontally because sessions need to be shared among servers (e.g., with a centralized session store).
  - **Use Case:** Traditional web applications where the server manages the session state.
- **Token-based Authentication (e.g., JWT):**
  - **Mechanism:** A token (like a JWT) containing user data is created on the server and sent to the client. The client stores this token and sends it with every request.
  - **State: Stateless**. The server doesn't need to store any session information. It only needs the secret key to verify the token's signature.
  - **Scalability:** Highly scalable because there is no server-side state. Any server can handle any request.
  - **Use Case:** APIs, mobile apps, and single-page applications where the backend is decoupled from the frontend.

33. What is Passport.js? How to use it with Express.js?

Passport.js is a popular and flexible authentication middleware for Node.js. It's a "passport" for your app, and it provides a clean, modular way to handle different authentication strategies (e.g., local login, Google OAuth, Facebook, JWT).

How to use:

1. **Install Passport.js and a strategy:** `npm install passport passport-local`
2. **Configure Express and Passport:**
3. JavaScript

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;
const expressSession = require('express-session');

app.use(expressSession({ secret: 'secret' }));
```

```
app.use(passport.initialize());
app.use(passport.session());
```

4.
5.
6. **Define a strategy and use it:**
7. JavaScript

```
passport.use(new LocalStrategy(
  (username, password, done) => {
    // Find user and verify password
    // If successful, call `done(null, user);`
    // If fail, call `done(null, false);`
  }
));
```

8.
9.
10. **Use Passport in your login route:**
11. JavaScript

```
app.post('/login', passport.authenticate('local', {
  successRedirect: '/',
  failureRedirect: '/login'
}));
```

12.
13.

34. How to implement JWT authentication in Express.js?

JSON Web Tokens (JWTs) are a secure way to transmit information between parties. The process:

1. **User logs in:** The server verifies the user's credentials.
2. **Create a JWT:** The server creates a token containing user-specific data (e.g., user ID, roles). It signs the token with a secret key.
3. **Send the token:** The server sends the JWT back to the client.
4. **Client stores the token:** The client stores the token (e.g., in `localStorage` or a cookie).
5. **Subsequent requests:** The client sends the token in the `Authorization` header with every request.
6. **Verify the token:** A middleware on the server verifies the token's signature using the same secret key. If the signature is valid, the request proceeds.

Example with `jsonwebtoken` and `express-jwt`:

1. **Install:** `npm install jsonwebtoken express-jwt`
2. **Login route:**
3. JavaScript

```
const jwt = require('jsonwebtoken');
app.post('/login', (req, res) => {
  // ... check credentials ...
  const token = jwt.sign({ userId: user.id }, 'my-secret-key', { expiresIn: '1h' });
  res.json({ token });
});
```

4.
5.
6. **Protect a route with middleware:**
7. JavaScript

```
const jwtAuthz = require('express-jwt');
app.use(jwtAuthz({ secret: 'my-secret-key', algorithms: ['HS256'] }));

app.get('/protected', (req, res) => {
  res.send('This is protected data!');
});
```

8.
9.

35. What is CSRF? How to prevent it in Express.js?

CSRF (Cross-Site Request Forgery) is an attack where a malicious website or email tricks a user's browser into making an unwanted request to a trusted site where the user is already authenticated.

How to prevent:

The most common and effective way is to use a CSRF token.

1. When a user requests a form (e.g., for a bank transfer), the server generates a unique, unpredictable token and includes it as a hidden field in the form. The token is also stored in the user's session.
2. When the user submits the form, the token is sent with the request.
3. The server verifies that the token in the request matches the one stored in the session.
4. If they don't match, the request is rejected.
   This works because the malicious site cannot access or predict the unique token stored in the user's session.

Express has middleware like `csurf` that automates this process.

36. How to prevent XSS in Express.js apps?

XSS (Cross-Site Scripting) is an attack where an attacker injects malicious scripts into a web page viewed by other users. This can lead to stealing cookies, session hijacking, etc.

Prevention:

1. **Input Validation and Sanitization:** Never trust user input. Sanitize all user-generated content before rendering it.
2. **Output Encoding/Escaping:** Escape all HTML, JavaScript, and other code before rendering it to the client. Modern templating engines like EJS and Pug do this by default.
3. **Content Security Policy (CSP):** Use the `helmet` middleware to set a CSP header. A CSP is a security policy that whitelists allowed sources of content (scripts, styles, etc.) for a web page.
4. **Use a library:** Use a library like `xss-filters` or `sanitize-html` to properly sanitize user input.

37. What is Helmet middleware? Why use it?

Helmet is a collection of 14 middleware functions that set various HTTP headers to help protect your Express.js application from common web vulnerabilities.

Why use it?

- It's a simple, set-it-and-forget-it solution for security headers.
- It protects against vulnerabilities like XSS, clickjacking, and header sniffing.
- It sets a Content Security Policy (CSP), hides the `X-Powered-By` header, and more.

Example:

```
JavaScript
```

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

app.use(helmet()); // This enables all of the default headers
```

38. How to use CORS in Express.js?

CORS (Cross-Origin Resource Sharing) is a browser security feature that prevents a web page from making requests to a different domain than the one that served the page. You need to enable CORS in your Express API if your frontend is hosted on a different domain.

How to enable:

1. **Install `cors` middleware:** `npm install cors`
2. **Use it globally:**
3. JavaScript

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

```
app.use(cors()); // This enables CORS for all routes
```

4.
5.
6. **Configure for specific origins:**
7. JavaScript

```
const corsOptions = {
  origin: 'https://www.example.com', // Only allow this origin
  optionsSuccessStatus: 200
};

app.get('/products/:id', cors(corsOptions), (req, res, next) => {
  res.json({ msg: 'This is CORS-enabled for a specific origin.' });
});
```

8.
9.

39. How to implement role-based access control (RBAC) in Express.js?

RBAC involves restricting access to certain parts of your application based on a user's role (e.g., `admin`, `editor`, `user`).

Implementation:

1. **Store roles:** The user's role should be stored in the database and included in the authentication token or session.
2. **Create a middleware function:** This function checks the user's role and grants or denies access.
3. JavaScript

```
const isAdmin = (req, res, next) => {
  if (req.user && req.user.role === 'admin') {
    next(); // User is an admin, proceed
  } else {
    res.status(403).send('Access Denied'); // Forbidden
  }
};
```

4.
5.
6. **Apply the middleware to routes:**
7. JavaScript

```
// First, ensure the user is authenticated
app.get('/admin/dashboard', isAuthenticated, isAdmin, (req, res) => {
  res.send('Welcome, Admin!');
});
```

8.
9. This is often used with libraries like `express-jwt-authz` for JWT-based RBAC.

40. How to protect routes in Express.js?

Protecting routes means ensuring that only authenticated or authorized users can access them.

Method: Use middleware.

1. **Authentication Middleware:** A middleware function checks for a valid session or token. If one isn't present, it sends an error response (e.g., `401 Unauthorized`) and terminates the request.
2. JavaScript

```
const authenticateToken = (req, res, next) => {
  const token = req.headers['authorization'];
  if (!token) return res.sendStatus(401);
  // ... verify token logic ...
  next();
};
```

3.
4.
5. **Apply the middleware:**
6. JavaScript

```
// All routes under /api/protected will require a valid token
app.use('/api/protected', authenticateToken);

// This route is now protected
app.get('/api/protected/data', (req, res) => {
  res.send('Secret data.');
});
```

7.
8.

This creates a firewall-like layer where unauthenticated requests are stopped before they reach the route handler.

---

5. Performance & Scaling

41. How to improve the performance of Express.js applications?

- **Use Middleware Judiciously:** Only use the middleware you need, and apply it locally where possible.

- **Implement Caching:** Cache frequently accessed data (e.g., with Redis or an in-memory cache) to reduce database queries.
- **Enable Gzip Compression:** Use `compression` middleware to reduce the size of responses.
- **Asynchronous Operations:** Use `async/await` and promises to handle I/O-bound tasks without blocking the event loop.
- **Clustering:** Use Node.js's built-in `cluster` module or a tool like PM2 to take advantage of multiple CPU cores.
- **Load Balancing:** Distribute incoming traffic across multiple instances of your application.
- **Use a CDN:** Serve static files from a Content Delivery Network.
- **Profile and Monitor:** Use tools like `New Relic` or `clinic.js` to identify performance bottlenecks.

42. What is clustering in Node/Express?

Clustering is the practice of running multiple instances of your Node.js application, with each instance running on a separate CPU core. Node.js is single-threaded, so a single instance can only utilize one core. By clustering, you can leverage all available cores on a server, significantly improving performance and handling more concurrent connections.

- **How it works:** A "master" process listens for incoming requests and distributes them to "worker" processes, which are the actual instances of your application.
- **Implementation:** You can use the built-in `cluster` module or a process manager like **PM2**, which automates this process and handles tasks like restarting failed workers.

43. How to implement caching in Express.js? (Redis, memory-cache).

Caching stores the result of an expensive operation (like a database query) so that subsequent requests for the same data can be served faster.

- **In-Memory Caching:** Simple and fast, but data is lost when the server restarts. A library like `memory-cache` is easy to use.
- JavaScript

```
const cache = require('memory-cache');
app.get('/data', (req, res) => {
  let data = cache.get('my-data');
  if (!data) {
    // Data not in cache, fetch it from DB
    // ...
    cache.put('my-data', data, 60000); // Cache for 1 minute
  }
  res.json(data);
});
```

-
-

- **External Caching (Redis):** A more robust solution where the cache is a separate, persistent service. This is ideal for a cluster of servers, as all instances can share the same cache.

44. What is compression middleware in Express.js?

The `compression` middleware is a Gzip compression library for Express. It compresses response bodies for all requests that pass through it. This significantly reduces the size of the data sent from the server to the client, leading to faster load times.

- **How to use:**
    1. **Install:** `npm install compression`
    2. **Use it:**
- JavaScript

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

app.use(compression());
```

-
-
- It should be placed early in the middleware stack, before any routes that send a response, to ensure all responses are compressed.

45. Difference between load balancing and clustering.

- **Clustering** is a technique for running multiple instances of a **single application** on a **single machine** to utilize multiple CPU cores. It's about horizontal scaling on a single server.
- **Load Balancing** is a technique for distributing incoming network traffic across **multiple servers** (or multiple clusters of servers). It's about distributing the load to prevent any single server from becoming a bottleneck.

Think of it this way: **Clustering** makes your application run on all cores of a single machine. **Load Balancing** makes your application run on all of your machines. They are often used together for maximum scalability.

46. How to handle large file uploads efficiently?

- **Use a stream:** Instead of buffering the entire file into memory before processing, use a stream-based approach. Multer supports this with its `busboy` dependency.
- **Chunking:** Break large files into smaller chunks on the client side and upload them one by one. The server then reassembles the chunks.
- **Dedicated file storage:** Offload file storage to a dedicated service like **AWS S3, Google Cloud Storage, or Cloudinary**. This frees up your server's resources and storage. Your Express server can simply receive the file and then upload it to the external service.

47. What is rate limiting in Express.js? How to implement it?

Rate limiting is a technique used to control the number of requests a user can make to your server within a certain time window. This prevents abuse, brute-force attacks, and DDoS attacks.

How to implement:

The most popular middleware is express-rate-limit.

1. **Install:** npm install express-rate-limit
2. **Configure and use:**
3. JavaScript

```
const rateLimit = require('express-rate-limit');
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // max 100 requests per IP per window
  message: 'Too many requests from this IP, please try again after 15 minutes.'
});

// Apply the limiter to all requests
app.use(limiter);

// Or to specific routes
app.get('/login', limiter, (req, res) => {
  // ...
});
```

4.
5.

48. How to use a reverse proxy (Nginx) with Express.js?

A **reverse proxy** like Nginx sits in front of your Express.js server and forwards requests to it. This is a best practice for production environments.

Benefits of using a reverse proxy:

- **Load Balancing:** Distribute traffic to multiple Express instances.
- **Security:** Hides your Express server's IP and ports, adds a security layer.
- **Static File Serving:** Nginx is highly optimized for serving static files, which can offload this task from your Express server.
- **SSL/TLS Termination:** Handle HTTPS traffic at the proxy level.

Configuration example (Nginx):

Nginx

```
server {
```

```
    listen 80;
    server_name yourdomain.com;

    location / {
        proxy_pass http://localhost:3000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

49. What are some best practices for scaling Express apps?

- **Statelessness:** Design your application to be stateless so that any request can be handled by any server instance. This makes it easy to add more servers.
- **Use a process manager:** Use PM2 to manage, monitor, and scale your application.
- **Externalize resources:** Store sessions in a separate database (like Redis) and upload files to external services (like S3).
- **Caching:** Implement caching at multiple levels (reverse proxy, database, in-memory).
- **Database Optimization:** Use indexes, optimize queries, and consider database sharding.
- **Monitoring:** Set up monitoring and alerting to detect performance issues early.

50. How to handle asynchronous operations in Express.js (async/await vs promises)?

Express.js is built for asynchronous I/O. Both `async/await` and promises are modern ways to handle this.

- **Promises:** Provide a structured way to handle asynchronous operations and chain them with `.then()` and `.catch()`.
- JavaScript

```
someAsyncFunction()
  .then(result => res.send(result))
  .catch(err => next(err));
```

-
-
- **async/await:** A syntactic sugar on top of promises that makes asynchronous code look and feel synchronous. It's generally more readable and easier to debug.
- JavaScript

```
app.get('/data', async (req, res, next) => {
  try {
```

```
    const data = await someAsyncFunction();
    res.json(data);
  } catch (err) {
    next(err); // Pass error to the error handler
  }
});
```

The async/await approach is now the standard for Express.js development as it avoids "callback hell" and handles errors gracefully with try...catch.

---

6. Error Handling & Logging

51. How does Express.js handle errors?

Express handles errors through a dedicated **error-handling middleware**. When an error occurs, either from an internal source or explicitly passed with next(err), Express bypasses the rest of the middleware stack and jumps to the first error-handling middleware it finds.

- **Synchronous errors:** Errors thrown synchronously are caught automatically by Express.
- **Asynchronous errors:** You must explicitly pass the error to next(), or use try...catch blocks with async/await.

52. What is centralized error handling in Express.js?

Centralized error handling is the practice of having a single, dedicated error-handling middleware function at the end of your middleware stack. This function catches all errors from your application, logs them, and sends a consistent, user-friendly error response. This prevents you from having to handle errors in every single route or middleware function.

Example:

```
JavaScript

// ... all your app.get(), app.post() etc. ...

// This is the centralized error handler
app.use((err, req, res, next) => {
  console.error(err.stack); // Log the error for debugging
  res.status(500).send('Internal Server Error');
});
```

53. Difference between operational errors and programmer errors.

- **Operational Errors:** These are predictable errors that are part of the normal operation of your application. They are non-critical and can be handled gracefully.

    - **Examples:** Invalid user input, a file not being found, network issues, database connection timeouts.
- **Programmer Errors:** These are bugs or logic errors in your code. They are unpredictable and indicate a fundamental flaw in the application. They are usually not recoverable and the best course of action is to crash the application and restart.
    - Examples: Calling a method on undefined, trying to read a property of null, syntax errors.
      It's crucial to handle operational errors gracefully and log programmer errors for debugging.

54. How to implement logging in Express.js? (Winston, Morgan).

- `morgan`: A simple HTTP request logger middleware. It's great for logging incoming requests and their status codes.
    - **Use case:** Development and basic production logging.
    - **Example:** `app.use(morgan('tiny'));`
- `winston`: A more advanced, robust logger that provides more control over log levels, transports (where logs are sent, e.g., to a file, database, or console), and formatting.
    - **Use case:** Production-level logging where you need to save logs to a file or a service.
    - **Example:**
- JavaScript

```
const winston = require('winston');
const logger = winston.createLogger({
  transports: [
    new winston.transports.Console(),
    new winston.transports.File({ filename: 'app.log' })
  ]
});
logger.info('User logged in successfully.');
```

-
-

55. How to debug an Express.js application?

- `console.log()`**:** The most basic method. Use it to print variable values and track the flow of your code.
- `node --inspect`**:** Node.js's built-in debugger. You can attach a debugger from Chrome DevTools or a code editor like VS Code. This allows you to set breakpoints, step through code, and inspect variables.
- **Libraries like** `debug`**:** A small utility that enables conditional debugging output. You can turn on/off debugging for specific parts of your application using environment variables.
- **Express-specific tools:** Use third-party tools or a robust logging system (like Winston) to provide a clear picture of what's happening.

56. How to send custom error responses?

You can send custom error responses from your centralized error-handling middleware based on the type of error.

Example:

JavaScript

```
app.use((err, req, res, next) => {
  if (err.name === 'ValidationError') {
    // Custom response for validation errors
    return res.status(400).json({ message: 'Validation failed', details: err.errors });
  }

  // Fallback for all other errors
  res.status(500).json({ message: 'Internal Server Error' });
});

// A route that might throw a validation error
app.post('/user', (req, res, next) => {
  if (!req.body.name) {
    const error = new Error('Name is required');
    error.name = 'ValidationError';
    error.errors = [{ field: 'name', message: 'Name cannot be empty' }];
    next(error);
  }
});
```

57. Difference between synchronous and asynchronous error handling.

- **Synchronous Error Handling:** For errors that occur in synchronous code. You can use a simple `throw` statement or pass the error directly to `next()`. Express will automatically catch `throw` errors and pass them to your error-handling middleware.
- JavaScript

```
app.get('/sync-error', (req, res, next) => {
  throw new Error('This is a synchronous error');
});
```

-
-
- **Asynchronous Error Handling:** For errors in asynchronous code (e.g., in a callback or a promise). Express won't automatically catch these. You **must** explicitly pass the error to `next()`.
- JavaScript

```
app.get('/async-error', (req, res, next) => {
```

```
  // Example with a callback
  fs.readFile('/nonexistent-file', (err, data) => {
    if (err) return next(err); // Must call next()
    res.send(data);
  });
});
```

-
- Using `async/await` with `try...catch` simplifies this by allowing you to handle asynchronous errors in a synchronous-looking way.

58. What is the role of `process.on("uncaughtException")` and `process.on("unhandledRejection")`?

These are Node.js-level event handlers for catching errors that are not handled by your application's `try...catch` blocks or Express's middleware.

- `process.on("uncaughtException")`: Catches synchronous errors that were not caught by a `try...catch` block.
- `process.on("unhandledRejection")`: Catches promise rejections that were not handled by a `.catch()` block.

Role: They are the last line of defense for catching critical, programmer-level errors. They are typically used to perform a **graceful shutdown** (e.g., log the error, close the database connection) and then exit the process. It's **not recommended** to continue the application after an `uncaughtException`, as the application state is likely corrupted.

59. How to implement a global error handler?

A global error handler is simply an error-handling middleware function that is placed **at the very end** of your middleware stack, **after all other route definitions and middleware**.

Example:

```
JavaScript

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

// ... all other middleware and routes ...

// The global error handler
app.use((err, req, res, next) => {
  console.error(err.stack); // Log the full stack trace for debugging
  res.status(500).send('Something broke!');
});
```

This ensures that any error, regardless of where it originates in your application, will eventually be caught and handled consistently.

60. How to handle database errors in Express.js?

Database errors are a form of operational error. You should catch them and handle them gracefully.

1. **Use `try...catch` with `async/await`:** This is the most common and readable way.
2. JavaScript

```
app.get('/users/:id', async (req, res, next) => {
  try {
    const user = await User.findById(req.params.id);
    if (!user) {
      return res.status(404).send('User not found.');
    }
    res.json(user);
  } catch (err) {
    // Pass the database error to the global error handler
    next(err);
  }
});
```

3.
4.
5. **Pass errors to `next()`:** If you're using callbacks or a library that doesn't support promises, you must manually pass the error to the `next` function.
6. **Use a global handler:** Your centralized error handler should be able to identify and handle database-specific errors (e.g., from Mongoose or Sequelize) and send an appropriate response.

---

7. Database & APIs

61. How to connect Express.js with MongoDB?

You use an **ODM (Object Data Modeling)** library like **Mongoose**. Mongoose provides a schema-based solution to model your application data, enforcing structure and providing a rich set of features for interacting with the database.

Steps:

1. **Install Mongoose:** `npm install mongoose`
2. **Connect to MongoDB:**
3. JavaScript

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/mydatabase', {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
```

```
.then(() => console.log('Connected to MongoDB!'))
.catch(err => console.error('Could not connect to MongoDB...', err));
```

4.
5.
6. **Define a Schema and Model:**
7. JavaScript

```
const userSchema = new mongoose.Schema({
  name: String,
  email: { type: String, required: true }
});
const User = mongoose.model('User', userSchema);
```

8.
9.
10. **Use the model in your routes:**
11. JavaScript

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

12.
13.

62. How to connect Express.js with SQL databases?

You use a database driver or an **ORM (Object-Relational Mapping)** library.

- **Without an ORM (using a driver like `pg` or `mysql2`):**
- JavaScript

```
const mysql = require('mysql2');
const connection = mysql.createConnection({
  host: 'localhost', user: 'root', database: 'test'
});
app.get('/users', (req, res) => {
  connection.query('SELECT * FROM users', (err, results) => {
    if (err) return res.status(500).send(err);
    res.json(results);
  });
});
```

-
-
- **With an ORM (like Sequelize):** Sequelize provides an abstraction layer that allows you to interact with the database using JavaScript objects, rather than raw SQL.
- JavaScript

```
const { Sequelize, DataTypes } = require('sequelize');
const sequelize = new Sequelize('database', 'username', 'password', { host: 'localhost', dialect:
'mysql' });
const User = sequelize.define('User', { name: DataTypes.STRING });
app.get('/users', async (req, res) => {
  const users = await User.findAll();
  res.json(users);
});
```

-
-

63. Difference between using ORM (Sequelize, TypeORM) vs ODM (Mongoose).

Aspect	ORM (Object-Relational Mapping)	ODM (Object-Document Mapping)
Database Type	Relational databases (SQL)	Document databases (NoSQL, like MongoDB)
Data Model	Mapped to tables and rows. Relationships are explicit (e.g., foreign keys).	Mapped to collections and documents. Relationships are often embedded or denormalized.
Primary Goal	Translate database-specific queries (SQL) into object-oriented code.	Provide a schema layer on top of a schemaless database.
Example Libraries	Sequelize, TypeORM, Prisma	Mongoose
Use Case	Ideal for applications with a rigid, structured data model and complex relationships.	Ideal for applications with flexible, dynamic data structures and high-speed write operations.

64. How to build REST APIs using Express.js?

Building a REST API with Express involves defining routes that correspond to the four main CRUD (Create, Read, Update, Delete) operations, using the appropriate HTTP verbs.

- **C (Create):** POST /api/resources to create a new resource.
- **R (Read):** GET /api/resources to get all resources, and GET /api/resources/:id to get a single resource.
- **U (Update):** PUT /api/resources/:id to update an existing resource.
- **D (Delete):** DELETE /api/resources/:id to delete a resource.

Example:

JavaScript

```
// GET all users
app.get('/api/users', async (req, res) => { /* ... */ });
// GET a single user
app.get('/api/users/:id', async (req, res) => { /* ... */ });
// POST to create a user
app.post('/api/users', async (req, res) => { /* ... */ });
// PUT to update a user
app.put('/api/users/:id', async (req, res) => { /* ... */ });
// DELETE a user
app.delete('/api/users/:id', async (req, res) => { /* ... */ });
```

65. How to implement GraphQL with Express.js?

GraphQL is a query language for your API. You define a single endpoint (/graphql) that handles all requests, and the client specifies exactly what data it needs. You can use the express-graphql middleware to quickly set up a GraphQL server.

Steps:

1. **Install:** npm install express-graphql graphql
2. **Define a schema:** Describe your data types and the queries/mutations available.
3. **Create a root resolver:** Functions that fetch the requested data.
4. **Use the middleware:**
5. JavaScript

```
const { graphqlHTTP } = require('express-graphql');
const schema = require('./schema'); // Your GraphQL schema
const root = require('./resolvers'); // Your root resolver

app.use('/graphql', graphqlHTTP({
  schema: schema,
  rootValue: root,
  graphiql: true, // Enable the in-browser IDE for testing
}));
```

6.
7. This sets up a GraphQL endpoint at /graphql.

66. How to handle pagination in Express.js API?

Pagination is crucial for APIs that return large lists of data. Instead of returning all records at once, you return them in chunks (pages). This is typically handled with query parameters.

Common approach: Use `req.query.page` and `req.query.limit` to control the pagination.

1. Get `page` and `limit` from `req.query`.
2. Calculate the `offset` (or `skip` for MongoDB): `offset = (page - 1) * limit`.
3. Use the `limit` and `offset` in your database query.

Example with Mongoose:

```
JavaScript


app.get('/posts', async (req, res) => {
  const page = parseInt(req.query.page) || 1;
  const limit = parseInt(req.query.limit) || 10;
  const skip = (page - 1) * limit;

  try {
    const posts = await Post.find().skip(skip).limit(limit);
    const totalCount = await Post.countDocuments();
    res.json({
      data: posts,
      total: totalCount,
      page,
      limit
    });
  } catch (err) {
    res.status(500).send(err);
  }
});
```

67. How to handle filtering and sorting in Express.js API?

Similar to pagination, filtering and sorting are handled with `req.query` parameters.

- **Filtering:** Use `req.query` parameters to build a dynamic query.
- JavaScript

```
app.get('/products', async (req, res) => {
  const filter = {};
  if (req.query.category) {
    filter.category = req.query.category;
  }
  if (req.query.price_min) {
    filter.price = { $gte: req.query.price_min };
  }
```

```
  const products = await Product.find(filter);
  res.json(products);
});
```

-
-
- **Sorting:** Use a `req.query` parameter (e.g., `sort_by`) to determine the sort order.
- JavaScript

```
app.get('/products', async (req, res) => {
  let sort = {};
  if (req.query.sort_by === 'price_asc') {
    sort.price = 1;
  } else if (req.query.sort_by === 'price_desc') {
    sort.price = -1;
  }
  const products = await Product.find().sort(sort);
  res.json(products);
});
```

-
-

68. Difference between synchronous and asynchronous database queries.

- **Synchronous queries** block the entire application's execution until the query is complete. Node.js's single-threaded nature means this is a bad practice, as it would make your application unresponsive to other requests.
- **Asynchronous queries** are the standard in Node.js. They don't block the event loop. The database operation runs in the background, and a **callback** or **promise** is used to handle the result once the query is finished.

All modern Node.js database drivers and ORMs (like Mongoose and Sequelize) use an asynchronous, non-blocking model by default.

69. What is API versioning in Express.js?

API versioning is the practice of maintaining multiple versions of your API to allow older clients to continue to function while you release updates for newer ones.

Common strategies:

- **URL-based versioning:** The version is included in the URL (e.g., `/api/v1/users`, `/api/v2/users`).
- **Header-based versioning:** The version is sent in a custom HTTP header (e.g., `Accept-Version: v2`).
- **Query parameter versioning:** The version is a query parameter (e.g., `/api/users?version=2`).

URL-based versioning is the most common and easiest to implement in Express. You can simply create different route files for each version.

70. How to handle rate limiting in APIs?

Handling rate limiting in APIs is essential to prevent abuse and ensure fair access. You can use the `express-rate-limit` middleware, which was discussed earlier. It works well for general API endpoints.

Common use cases:

- **Limit sign-up and login endpoints:** Prevent brute-force attacks.
- **Limit resource-intensive endpoints:** For example, an endpoint that performs a complex calculation or returns a large amount of data.
- **Limit all API calls:** Apply a global rate limit to prevent DDoS attacks.
- **Custom limits for different user types:** Allow premium users to have a higher request limit.

---

8. Testing in Express.js

71. How to test Express.js applications?

Testing Express apps involves testing different layers:

- **Unit testing:** Testing individual functions or modules in isolation (e.g., a helper function, a model method).
- **Integration testing:** Testing the interaction between different components (e.g., a route handler and the database).
- **End-to-end (E2E) testing:** Simulating a user's journey through the entire application (e.g., login, navigate to a page, submit a form).

Tools:

- **Test Frameworks:** Mocha, Jest
- **Assertion Libraries:** Chai, Jest's built-in `expect`
- **Request Super-agents:** Supertest (for making HTTP requests to your Express app in a test)
- **Mocking Libraries:** Sinon, Jest's `jest.mock`

72. Difference between unit testing and integration testing in Express.js.

- **Unit Testing:** Focuses on a single unit of code, like a function or a class. You mock all external dependencies (e.g., the database, other services) to ensure you are only testing the unit's logic.
- **Integration Testing:** Tests how different components work together. For an Express app, this often means testing a route handler's full lifecycle: from receiving the HTTP request to making a database call and sending the final response. You're testing the "integration" of your route, middleware, and database.

73. How to mock requests and responses in testing?

When unit-testing a middleware or a route handler, you don't want to spin up a full server. You can create mock objects for the `req` and `res` objects. A library like `sinon` or `jest.mock` can help.

Example with Jest:

```
JavaScript

// Test a simple route handler
test('should return "Hello World!"', () => {
  const req = {}; // Mock request object
  const res = {
    send: jest.fn() // Mock the res.send() method
  };

  myRouteHandler(req, res);

  // Assert that res.send was called with the correct argument
  expect(res.send).toHaveBeenCalledWith('Hello World!');
});
```

However, this method is more suited for unit tests. For integration tests, **Supertest** is the preferred solution.

74. What is Supertest? How to use it with Mocha/Jest?

Supertest is a popular library for testing HTTP requests. It allows you to make "fake" HTTP requests to your Express application without having to listen on a real port. It's perfect for integration testing.

How to use with Jest:

1. **Install:** `npm install supertest --save-dev`
2. **Write a test:**
3. ```
   JavaScript


   const request = require('supertest');
   const app = require('./app'); // Your main Express app file

   describe('GET /', () => {
     test('should respond with "Hello, World!"', async () => {
       const response = await request(app).get('/');
       expect(response.statusCode).toBe(200);
       expect(response.text).toBe('Hello, World!');
     });
   });
   ```
4.
   ```

5.

Supertest handles creating a temporary server and making the requests, making your tests clean and efficient.

75. What is Chai and Sinon in testing Express.js apps?

- **Chai:** An assertion library. It provides a readable, fluent API for making assertions.
  - **Styles:** `expect` (`expect(foo).to.be.a('string')`), `should`, `assert`.
  - **Use case:** Used to verify that your code behaves as expected.
- **Sinon:** A powerful library for mocking, spying, and stubbing.
  - **Spy:** A function that records arguments, return values, etc. without changing the behavior of the original function.
  - **Stub:** Replaces a function with a new, controlled function.
  - **Mock:** A complete mock object with predefined behavior.
  - **Use case:** To isolate units of code by replacing dependencies with controlled test doubles. For example, stubbing a database call to return a specific value.

76. How to test middleware in Express.js?

Testing middleware is similar to testing a regular function. You need to mock the `req`, `res`, and `next` objects.

Example with Jest:

JavaScript

```
const myMiddleware = require('./middleware/myMiddleware');

test('myMiddleware should add a timestamp to the request', () => {
  const req = {};
  const res = {};
  const next = jest.fn(); // Mock the next() function

  myMiddleware(req, res, next);

  expect(req.timestamp).toBeDefined();
  expect(next).toHaveBeenCalledTimes(1);
});
```

77. How to test routes in Express.js?

Testing routes is a form of integration testing and is best done with **Supertest**. It allows you to send real HTTP requests to your application and assert on the response.

Example:

JavaScript

```
const request = require('supertest');
const app = require('./app');

describe('POST /api/users', () => {
  test('should create a new user', async () => {
    const newUser = { name: 'Alice' };
    const response = await request(app)
      .post('/api/users')
      .send(newUser)
      .expect(201); // Assert the status code

    expect(response.body.name).toBe('Alice');
  });
});
```

This test sends a POST request to a route, sends a JSON body, and asserts that the response is correct.

78. How to test error handling?

You can test error-handling middleware by making a request that will intentionally cause an error.

Example:

```
JavaScript

const request = require('supertest');
const app = require('./app');

describe('GET /broken-route', () => {
  test('should return a 500 status code', async () => {
    await request(app).get('/broken-route').expect(500);
  });
});
```

This works because the Supertest call will trigger the route, which throws an error, which is then caught by your error-handling middleware, which sends a 500 status code. The test then asserts that the status code is correct.

79. What is end-to-end testing in Express.js (Cypress)?

End-to-end (E2E) testing simulates the full user experience from start to finish. It tests the frontend, the API, and the database as a complete system. Tools like **Cypress** or **Playwright** automate a browser to click buttons, fill out forms, and navigate pages, while your Express server is running on a local port.

Use Case: Ensures that a critical user flow (e.g., user signup, login, and profile update) works correctly in a production-like environment. It's slower and more complex than unit/integration testing but provides a higher degree of confidence.

80. Best practices for testing Express.js applications.

- **Start with Integration Tests:** Integration tests using Supertest provide the most value for Express apps. They catch errors from middleware, routes, and database interactions.
- **Write Tests for Critical Logic:** Focus on testing core business logic and critical user flows.
- **Use a CI/CD Pipeline:** Automate your tests to run on every code change.
- **Use a Mocking Library:** Use libraries like Sinon or Jest's built-in mocks to isolate units and control external dependencies.
- **Test All Layers:** Use a mix of unit, integration, and E2E tests for comprehensive coverage.
- **Test for Edge Cases:** Test for invalid input, missing parameters, and unexpected errors.

---

9. Advanced Express.js Concepts

81. Difference between REST API and GraphQL API in Express.js.

Aspect	REST API	GraphQL API
Endpoints	Multiple endpoints, each for a specific resource (`/users`, `/products/1`).	A single endpoint (`/graphql`) for all requests.
Request Model	Uses HTTP verbs (GET, POST, PUT, DELETE) and URLs to define operations.	Uses a query language to describe the data needed. All requests are typically a POST.
Data Fetching	Client receives all or a predetermined set of data for a resource. This can lead to **over-fetching** (getting too much data) or **under-fetching** (needing multiple requests).	Client specifies exactly what data it needs. No over-fetching or under-fetching.

Use Case	Traditional web apps, when resources are clearly defined.	Complex applications, mobile apps, or when the client needs to fetch related data in a single request.

82. What is API Gateway? How to use it with Express.js?

An **API Gateway** is a server that acts as a single entry point for all API requests. It handles tasks like authentication, routing, rate limiting, and caching before forwarding the request to the appropriate microservice.

How to use with Express:

You would use an API Gateway (like AWS API Gateway or Kong) in front of your Express.js microservices. The gateway would receive all incoming requests, and based on the path, it would route the request to the correct Express service. This allows you to centralize cross-cutting concerns (like authentication) at the gateway level.

83. What are microservices? How to implement with Express.js?

Microservices are an architectural style where a large application is built as a collection of small, independent services that communicate with each other. Each service is responsible for a single business capability.

How to implement with Express:

You would build each microservice as a separate Express.js application. For example, one Express app for a User service, another for an Order service, and a third for a Payment service. Each service would have its own database and can be deployed and scaled independently. You would then use an API Gateway to route traffic to them.

84. What is service discovery in microservices architecture?

In a microservices architecture, services need to be able to find and communicate with each other. **Service discovery** is the process by which services locate other services on the network.

How it works:

1. When a service starts, it registers itself with a **service registry** (e.g., Eureka, Consul).
2. When a service needs to call another service, it queries the service registry to find its location (IP and port).
3. The service registry returns the location, and the calling service can make the request.
   This prevents you from having to hardcode the location of services, making your architecture more flexible and resilient.

85. Difference between monolithic and microservices architecture in Express.js context.

Aspect	Monolithic Architecture	Microservices Architecture
Structure	A single, large application that contains all components (routes, business logic, UI).	A collection of small, independent services.
Deployment	Deployed as a single unit.	Each service is deployed independently.
Scaling	Must scale the entire application, even if only one part is the bottleneck.	Can scale individual services independently.
Technology	All components use the same technology stack.	Services can be built with different technologies (e.g., Express.js for one, Python for another).
Complexity	Simpler to develop and deploy initially.	More complex to manage and operate due to distributed nature.

86. How to use Express.js with Docker?

Docker is a tool that packages an application and its dependencies into a container, ensuring it runs consistently in any environment.

Steps:

1. **Create a `Dockerfile`:** A text file that contains instructions for building a Docker image for your Express app.
2. Dockerfile

```
FROM node:18
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
```

```
EXPOSE 3000
CMD ["node", "app.js"]
```

3.
4.
5. **Build the image:** `docker build -t my-express-app .`
6. Run the container: `docker run -p 3000:3000 my-express-app`
   This makes your Express app portable and easy to deploy.

87. How to use Express.js with Kubernetes?

Kubernetes is an open-source system for automating the deployment, scaling, and management of containerized applications.

Steps:

1. **Containerize your Express app** with a Docker image (as above).
2. **Create a Deployment file:** A YAML file that tells Kubernetes how to run your application (e.g., how many replicas, what image to use).
3. **Create a Service file:** A YAML file that exposes your application to the outside world.
4. **Deploy to the cluster:** `kubectl apply -f deployment.yaml`

Kubernetes will automatically manage your Express containers, ensuring they are always running, load-balancing traffic, and scaling them as needed.

88. How to implement WebSockets with Express.js?

WebSockets are a protocol that provides full-duplex, two-way communication channels over a single TCP connection. This is ideal for real-time applications like chat apps or live dashboards.

Implementation: You need a library like `socket.io` or `ws`. Express doesn't have native support for WebSockets.

Example with `socket.io`:

1. **Install:** `npm install socket.io`
2. **Attach `socket.io` to your Express server:**
3. JavaScript

```
const express = require('express');
const http = require('http');
const socketio = require('socket.io');

const app = express();
const server = http.createServer(app);
const io = socketio(server);

io.on('connection', (socket) => {
  console.log('A user connected.');
```

```
  socket.on('chat message', (msg) => {
    io.emit('chat message', msg); // Broadcast the message
  });
});

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

4.

5.

89. Difference between WebSockets and REST APIs in Express apps.

Aspect	WebSockets	REST APIs
Communication	Two-way (full-duplex) communication. The server can push data to the client at any time.	One-way (client-initiated) communication. The client must make a request to get data.
Protocol	A separate protocol built on top of TCP.	Uses the standard HTTP protocol.
State	**Stateful**. A persistent connection is maintained.	**Stateless**. Each request is independent.
Use Case	Real-time applications (chat, gaming, live updates).	Standard CRUD operations, fetching data for a traditional web page.

90. How to implement Server-Sent Events (SSE) in Express.js?

Server-Sent Events (SSE) is a standard that allows a server to push data to a client over a single, long-lived HTTP connection. Unlike WebSockets, it's a one-way communication channel (server to client only).

Implementation:

- Set the `Content-Type` header to `text/event-stream`.
- Send data to the client with a specific format (`data: ...`).
- End the connection when done.

Example:

```
JavaScript


app.get('/events', (req, res) => {
  res.setHeader('Content-Type', 'text/event-stream');
  res.setHeader('Cache-Control', 'no-cache');
  res.setHeader('Connection', 'keep-alive');
  res.flushHeaders();

  let counter = 0;
  const interval = setInterval(() => {
    counter++;
    res.write(`id: ${counter}\n`);
    res.write(`data: The current time is ${Date.now()}\n\n`);
  }, 1000);

  req.on('close', () => {
    clearInterval(interval);
  });
});
```

---

10. Deployment & Real-World

91. How to deploy Express.js app on Heroku?

Heroku is a platform-as-a-service (PaaS) that makes deployment easy.

Steps:

1. **Create a Heroku app:** `heroku create my-app-name`
2. **Add a `Procfile`:** A file that tells Heroku how to run your app.

```
web: node app.js
```

3.
4.
5. **Set the port:** Heroku assigns a dynamic port. You must use `process.env.PORT` in your Express app.
6.

```
JavaScript


const port = process.env.PORT || 3000;
app.listen(port, ...);
```

7.
8.
9. **Deploy with Git:** `git push heroku main`

Heroku automatically installs dependencies and runs your application.

92. How to deploy Express.js app on AWS (EC2, Lambda)?

- **AWS EC2 (Elastic Compute Cloud):** A virtual server.
    - Launch an EC2 instance.
    - SSH into the instance and install Node.js.
    - Transfer your code (e.g., with `scp` or `git pull`).
    - Install dependencies (`npm install`).
    - Use a process manager like **PM2** to keep your app running.
- **AWS Lambda (Serverless):** A serverless function service.
    - Wrap your Express app in a serverless framework like `serverless-http` or `aws-serverless-express`.
    - Use the Serverless Framework or AWS SAM to deploy your function.
    - Configure an **API Gateway** to trigger the Lambda function when a request comes in.
    - **Benefit:** You only pay for what you use, and you don't have to manage servers.

93. How to deploy Express.js app on Docker?

1. **Create a `Dockerfile`** for your application.
2. **Build your Docker image:** `docker build -t my-app .`
3. **Push the image to a registry** (e.g., Docker Hub): `docker push your-repo/my-app:1.0`
4. **Deploy the container** to any environment that supports Docker (e.g., a cloud provider's managed container service, a self-managed server, or a Kubernetes cluster).

94. Difference between PM2 and nodemon.

- **`nodemon`**: A development tool. It monitors your source code and **automatically restarts** the server when it detects changes. It's for developer convenience.
- **`PM2`**: A production process manager. It keeps your application running 24/7. It handles:
    - **Clustering:** Spawning and managing multiple processes to use all CPU cores.
    - **Monitoring:** Provides a dashboard to monitor your app's performance.
    - **Automatic restarts:** Restarts your application if it crashes.
    - **Logging:** Manages logs for your application.

You use `nodemon` during development and `PM2` for production deployments.

95. How to configure environment variables in Express.js?

Environment variables are used to store configuration data (like database credentials, API keys, or ports) that change between different environments (development, production).

How to use:

- **Access them:** In your code, you access them using `process.env.VARIABLE_NAME`.
- **Set them (Linux/macOS):** `export VARIABLE_NAME=value` before running your app.
- **Set them (Windows):** `set VARIABLE_NAME=value`
- **Use a library:** Use the **`dotenv`** library for development.

96. What is dotenv in Express.js?

`dotenv` is a zero-dependency module that loads environment variables from a `.env` file into `process.env`. It's a standard practice for managing environment variables in development.

How to use:

1. **Install:** `npm install dotenv`
2. **Create a `.env` file:**

```
PORT=4000
DATABASE_URL=mongodb://localhost:27017/my-dev-db
```

3.
4.
5. **Require it at the top of your main file:**
6. JavaScript

```
require('dotenv').config();
const port = process.env.PORT || 3000;
```

7.
8.

`dotenv` is only for development. In production, you should set environment variables directly on the server or in your hosting provider's configuration.

97. How to implement CI/CD for Express.js?

CI/CD (Continuous Integration/Continuous Deployment) is a practice that automates the build, test, and deployment of your code.

Process:

1. **Continuous Integration (CI):**
    - You push code to your repository (e.g., GitHub).
    - A CI service (e.g., GitHub Actions, Jenkins) is triggered.
    - The service installs dependencies and runs your tests.
    - If the tests pass, a build is created (e.g., a Docker image).
2. **Continuous Deployment (CD):**
    - The build is automatically deployed to your staging or production environment.
    - Tools like AWS CodeDeploy, Heroku, or Kubernetes handle the deployment.

98. Difference between development and production mode in Express.js.

- **Development Mode:**
  - `NODE_ENV`: `development`
  - **Features:** Detailed error messages, live reloading (`nodemon`), verbose logging (`morgan('dev')`).
  - **Purpose:** Fast development, easy debugging.
- **Production Mode:**
  - `NODE_ENV`: `production`
  - **Features:** Caching, minified resources, less verbose error messages, disabled debugging tools, security headers (`Helmet`).
  - **Purpose:** Performance, security, and stability.

You should always set `NODE_ENV=production` when deploying your application to ensure it runs with optimized settings.

99. Common performance bottlenecks in Express apps and how to fix them.

- **Synchronous Operations:** Blocking the event loop with synchronous code (e.g., `fs.readFileSync`).
  - **Fix:** Use asynchronous APIs (`fs.readFile`) and `async/await`.
- **Inefficient Database Queries:** Full table scans, missing indexes.
  - **Fix:** Add indexes to frequently queried fields, optimize queries, and use a tool to profile slow queries.
- **Over-fetching Data:** Returning more data from an API than the client needs.
  - **Fix:** Use GraphQL or implement a filtering/field selection mechanism.
- **Unoptimized Middleware:** Using too many middleware functions or placing expensive middleware at the top of the stack.
  - **Fix:** Only use necessary middleware and use local middleware where possible.
- **Lack of Caching:** Repeatedly fetching the same data.
  - **Fix:** Implement caching with Redis or an in-memory cache.
- **Not Using All CPU Cores:** Running a single-instance Express app on a multi-core server.
  - **Fix:** Use **PM2** to run your app in a cluster.

100. Best practices for building secure, scalable Express.js applications.

- **Security:**
  - Use `helmet` middleware.
  - Implement CSRF protection.
  - Sanitize user input to prevent XSS.
  - Use strong password hashing (`bcrypt`).
  - Implement rate limiting.
  - Use HTTPS in production.
- **Scalability:**

    - Use a process manager like **PM2** for clustering.
    - Design a **stateless** API.
    - Use a reverse proxy (Nginx).
    - Externalize session storage and file uploads.
    - Implement caching.
- **Maintainability:**
    - Use modular routing with `express.Router()`.
    - Use `dotenv` for configuration.
    - Implement a centralized error handler.
    - Write comprehensive tests.
    - Follow a consistent coding style.