Intro to Client-Server Architecture (CSA)

CSA is a computing model where the workload is divided between providers of a resource or service, called servers, and service requesters, called clients.

Key Components

1. **Client:**

   - Requests services or resources.

   - Can be a web browser, mobile app, etc.

2. **Server:**

   - Provides services or resources.

   - Processes client requests and sends back responses.

How Requests are Handled at the Server

1. **Client Request:**

The client sends a request to the server, typically using HTTP/HTTPS.

The request contains information like the type of request (GET, POST, etc.), headers, and sometimes data.

2. **Server Processing:**

The server receives the request and processes it.

It may involve querying a database, performing computations, or other processing tasks.

3. **Response:**

The server sends a response back to the client.

The response includes status information (e.g., success or error) and any requested data.

Understanding Node.js

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine.

It allows developers to use JavaScript to write server-side code.

Advantages:

1. **Asynchronous and Event-Driven:**

Non-blocking I/O operations allow for handling multiple requests simultaneously, making it efficient for I/O-heavy tasks.

2. **Single Programming Language:**

Developers can use JavaScript for both client-side and server-side development, leading to consistency in code.

3. **Rich Ecosystem:**

NPM (Node Package Manager) provides a vast library of modules and packages.

4. Scalability:

Suitable for microservices and scalable applications due to its event-driven architecture.

5. **Performance:**

High performance for real-time applications like chat applications, games, etc.

Disadvantages:

1. **Single-Threaded Limitations:**

Not suitable for CPU-intensive tasks as it can lead to performance bottlenecks.

2. **Callback Hell:**

Asynchronous nature can lead to complex nested callbacks, making code difficult to read and maintain (though this can be mitigated with Promises and async/await).

3. **Maturity:** (Immature)

Some libraries and tools may not be as mature as those available in other languages like Java or Python.

Installing Node.js and Creating a Node.js Server

Step-by-Step Guide:

1. Install Node.js:

Download and install Node.js from the official website [Node.js](https://nodejs.org/).

Verify installation: node -v and npm -v

2. Create a Node.js Server:

Create a new directory for your project:

mkdir my-node-server , cd my-node-server

Initialize a new Node.js project: npm init -y

Install Express.js (a popular Node.js framework): npm install express

Create a server file `server.js`:

```js
const express = require('express');
const app = express();
const port = 3000;
app.get('/', (req, res) => {
  res.send('Hello World!');
});
app.listen(port, () => {
  console.log(`Server is running at http://localhost:${port}`);
});
```

Run the server: node server.js

Open a web browser and navigate to `http://localhost:3000`. You should see "Hello World!" displayed on the page.

Summary

Client-Server Architecture:

Divides workload between clients and servers.

Clients send requests, servers process and respond.

Node.js:

JavaScript runtime for server-side programming.

Asynchronous, event-driven, single language for full-stack development.

Pros: Efficiency, rich ecosystem, scalability, performance.

Cons: Single-threaded, potential callback hell, library maturity.

Creating a Node.js Server:

Install Node.js and verify. Initialize project and install Express.js.

Create a basic server script. Run and test the server.

Handling Requests(HR), Creating Endpoints(CE) , and Modules in Node.js

HR and CE

In Node.js, you handle HTTP requests and create endpoints typically using a framework like Express.js.

Here’s how you can handle different types of requests and create endpoints:

1. **GET Request:**

```js
app.get('/endpoint', (req, res) => {
  // Logic to handle GET request
  res.send('GET request to the homepage');
});
```

2. **POST Request:**

```js
app.post('/endpoint', (req, res) => {
  // Logic to handle POST request
  res.send('POST request to the homepage');
});
```

3. **PUT Request:**

~~~
```js
app.put('/endpoint', (req, res) => {
  // Logic to handle PUT request
  res.send('PUT request to the homepage');
});```
~~~

4. **DELETE Request:**

~~~
```js
app.delete('/endpoint', (req, res) => {
  // Logic to handle DELETE request
  res.send('DELETE request to the homepage');
});```
~~~

Modules in Node.js

Modules are reusable blocks of code that can be imported into other modules or files. They help in organizing code logically.

Built-in Modules

Node.js comes with several built-in modules like `http`, `fs`, `path`, etc.

http:

~~~
```js
const http = require('http');
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});
server.listen(3000, '127.0.0.1', () => {
  console.log('Server running at http://127.0.0.1:3000/');
~~~

```
});
```

fs:

```js
const fs = require('fs');
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

Creating and Exporting Custom Modules

1. Creating a Module (`myModule.js`):

```js
const myFunction = () => {
  console.log('Hello from my module!');
};
module.exports = myFunction;
```

2. **Importing and Using a Module:**

```js
const myFunction = require('./myModule');
myFunction(); // Output: Hello from my module!
```

npm (Node Package Manager)

npm is a package manager for Node.js packages.

It helps in installing, updating, and managing dependencies.

Installing a Package: npm install package-name

Uninstalling a Package: npm uninstall package-name

Updating Packages: npm update

Listing Installed Packages: npm list

Importing Modules

Modules can be imported using `require` or `import` syntax (ES6 modules).

Common JS (require): const express = require('express');

ES6 Modules (import): import express from 'express';

Handling Static Pages with File Stream

Serving static files (like HTML, CSS, JS) can be done using the `fs` module or middleware like `express.static`.

Using `fs` Module

```js
const fs = require('fs');
const http = require('http');
http.createServer((req, res) => {
  if (req.url === '/') {
    fs.readFile('index.html', (err, data) => {
      if (err) {
        res.writeHead(404, {'Content-Type': 'text/html'});
        res.end('404 Not Found');
      } else {
        res.writeHead(200, {'Content-Type': 'text/html'});
        res.end(data);
      }
    });
  }
}).listen(3000);
```

Using `express.static`

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

const port = 3000;

app.use(express.static('public'));

app.listen(port, () => {

  console.log(`Server is running at http://localhost:${port}`);

});
```

Handling Exceptions

Exception handling is crucial to ensure that the application does not crash and provides meaningful error messages.

Using try-catch

```js
try {

  // Code that may throw an error

} catch (error) {

  console.error(error);

}
```

Express Error Handling Middleware

```js
app.use((err, req, res, next) => {

  console.error(err.stack);

  res.status(500).send('Something broke!');

});
```

Node.js Frameworks

Node.js has several frameworks that simplify web development.

Here are some popular ones:

1. **Express.js:**

Lightweight and flexible. Middleware support and routing.

Widely used for building RESTful APIs.

2. **Koa.js:**

Created by the same team behind Express.

Uses async/await for better error handling. Minimalistic and modular.

3**. Hapi.js:**

Rich framework for building applications and services.

Focuses on configuration-driven development. Extensive plugin system.

4. **Nest.js:**

A progressive Node.js framework. Uses TypeScript.

Inspired by Angular, making it suitable for enterprise applications.

5. **Sails.js:**

MVC framework for Node.js. Similar to Ruby on Rails.

Supports data-driven APIs.

Summary

Handling Requests and Creating Endpoints:

Use frameworks like Express.js.

Different HTTP methods (GET, POST, PUT, DELETE).

Modules in Node.js:

Built-in modules (http, fs). Custom modules creation and usage.

npm:

Package installation, uninstallation, and updates. Dependency management.

Importing Modules: CommonJS (`require`) and ES6 Modules (`import`).

Handling Static Pages: Use `fs` module or `express.static`.

Handling Exceptions:

Use try-catch blocks. Express error handling middleware.

Node.js Frameworks: Express.js, Koa.js, Hapi.js, Nest.js, Sails.js.

Intro to Express

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

It is often used to build RESTful APIs and handle HTTP requests.

Key Features:

1. **Middleware:** Functions that execute during the lifecycle of a request to the server.

2. **Routing:** Defines how an application responds to a client request to a particular endpoint.

3. **Templating:** Dynamically render HTML pages.

4. **Static Files:** Serve static assets like HTML, CSS, JavaScript, images, etc.

5. **Robust API:** Supports various methods to interact with the HTTP protocol.

Serving Static Files

Express provides a built-in middleware function `express.static` to serve static files.

Example:

1. Create a folder named `public` and place your static files (e.g., `index.html`, `styles.css`, `script.js`) inside it.

2. Use the `express.static` middleware to serve the static files.

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

const port = 3000;

app.use(express.static('public'));

app.listen(port, () => {

  console.log(`Server is running at http://localhost:${port}`);

});
```

Routing in Express

Routing defines how the application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, etc.).

Routing Methods

GET: Retrieve data from the server.

POST: Send data to the server.

PUT: Update existing data on the server.

DELETE: Delete data from the server.

Example

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

const app = express();

const port = 3000;

app.get('/', (req, res) => {

  res.send('GET request to the homepage');

});

app.post('/', (req, res) => {

  res.send('POST request to the homepage');

});

app.put('/user', (req, res) => {

  res.send('PUT request to /user');
```

});

app.delete('/user', (req, res) => {

  res.send('DELETE request to /user');

});

app.listen(port, () => {

  console.log(`Server is running at http://localhost:${port}`);

});```

Route Paths

Define the endpoints of your application using strings, string patterns, or regular expressions.

Examples:

String: `/about`

String Pattern: `/ab?cd` (matches `/acd` and `/abcd`)

Regular Expression: `/a/` (matches any route that contains "a")

Route Parameters

Route parameters are named URL segments that are used to capture the values specified at their position in the URL.

Example

~~~js

app.get('/users/:userId/books/:bookId', (req, res) => {

  res.send(req.params);

});```

If you navigate to `/users/34/books/8989`, the response will be:

```json

{

  "userId": "34",

  "bookId": "8989"
~~~

```
}
```

Route Handlers

Multiple callback functions can handle a route. They are executed sequentially.

Example:

```js
app.get('/example', (req, res, next) => {
  console.log('First handler');
  next();
}, (req, res) => {
  res.send('Second handler');
});
```

Response Methods

Express provides methods to send a response to the client.

1. **res.send( ):** Sends a response of various types.

   ```js
   res.send('Hello World');
   ```

2. **res.json( ):** Sends a JSON response.

   ```js
   res.json({ message: 'Hello World' });
   ```

3. **res.sendFile( ):** Sends a file as an octet stream.

   ```js
   res.sendFile('/path/to/file');
   ```

4. **res.status( ):** Sets the HTTP status code.

```js
res.status(404).send('Not Found');
```

Summary

Intro to Express:

Minimal, flexible Node.js framework for web and mobile applications.

Features: Middleware, routing, templating, static files, robust API.

Serving Static Files:

Use `express.static` middleware.

Example provided for serving files from the `public` directory.

Routing:

Methods: GET, POST, PUT, DELETE.

Define routes using strings, patterns, or regular expressions.

Route parameters to capture values in the URL.

Multiple route handlers for a single route.

Response Methods:

`res.send()`: Sends a response.

`res.json()`: Sends a JSON response.

`res.sendFile()`: Sends a file.

`res.status()`: Sets the HTTP status code.

Middleware in Express

Middleware functions are functions that have access to the request object (`req`), the response object (`res`), and the next middleware function in the application’s request-response cycle.

These functions can execute code, modify the request and response objects, end the request-response cycle, and call the next middleware function in the stack.

Middleware Lifecycle

1. **Request Received:** When a request is received, it travels through a series of middleware functions before reaching the route handler.

2. **Execution of Middleware Functions:** Each middleware function can perform operations on the request or response objects, or end the request-response cycle.

3. **Calling Next Middleware:** If a middleware function calls `next()`, the next middleware function in the stack is executed.

4. **Route Handler Execution:** Once all middleware functions have executed, the request reaches the route handler which sends back the response.

Types of Middleware

1. **Application-level Middleware:**

Bound to an instance of the `express` object using `app.use()` or `app.METHOD()`, where `METHOD` is an HTTP method (e.g., `get`, `post`).

2. **Router-level Middleware:**

Works in the same way as application-level middleware but is bound to an instance of `express.Router()`.

3. **Error-handling Middleware:**

Defined with four arguments (err, req, res, next). It is used to handle errors that occur during request processing.

4. **Built-in Middleware:**

Middleware functions that come with Express, like `express.static`, `express.json`, and `express.urlencoded`.

5. **Third-party Middleware:** Middleware functions created by the community and available via npm, like `body-parser`, `morgan`, `cookie-parser`, etc.

Application-level Middleware

```js
const express = require('express');
const app = express();
// Middleware function
app.use((req, res, next) => {
  console.log('Time:', Date.now());
  next();
});
// Route handler
app.get('/', (req, res) => {
  res.send('Hello World');
});
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Router-level Middleware

```js
const express = require('express');
const app = express();
const router = express.Router();
// Middleware function
router.use((req, res, next) => {
  console.log('Request URL:', req.originalUrl);
  next();
});
// Route handler
router.get('/', (req, res) => {
  res.send('Router-level middleware');
});
app.use('/router', router);
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Error-handling Middleware

```js
const express = require('express');
const app = express();
// Middleware function
app.use((req, res, next) => {
  const err = new Error('Something went wrong!');
  err.status = 500;
  next(err);
});
// Error-handling middleware
app.use((err, req, res, next) => {
  res.status(err.status || 500);
  res.send({ error: err.message });
});
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Third-party Middleware

1. **Body-parser:** Parses incoming request bodies in a middleware before your handlers.

   ```js
   ```

````
   const bodyParser = require('body-parser');
   const express = require('express');
   const app = express();
   app.use(bodyParser.json()); // For parsing application/json
   app.use(bodyParser.urlencoded({ extended: true })); // For parsing
application/x-www-form-urlencoded
   app.post('/', (req, res) => {
     res.send(req.body);
   });
   app.listen(3000, () => {
     console.log('Server is running on port 3000');
   });```
````

2. **morgan:** HTTP request logger middleware for Node.js.

~~~
   ```js
   const morgan = require('morgan');
   const express = require('express');
   const app = express();
   app.use(morgan('dev'));
   app.get('/', (req, res) => {
     res.send('Hello World');
   });
   app.listen(3000, () => {
     console.log('Server is running on port 3000');
   });```
~~~

How Request Travels in Express

1. **Client Sends Request:** A client sends an HTTP request to the server.
2. **Middleware Stack:**

The request passes through a stack of middleware functions, which can modify the request and response objects or terminate the request-response cycle.

3. **Route Handler:**

If no middleware function terminates the cycle, the request reaches the route handler.

4. **Response Sent:**

The route handler processes the request and sends a response back to the client.

Blocking vs Non-blocking Code

Blocking Code

Synchronous:

Operations are executed sequentially.

Each operation must complete before the next one starts.

Example:

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

const data = fs.readFileSync('/file.md'); // Blocking

console.log(data);

console.log('This message is logged after reading the file');
```

Non-blocking Code

Asynchronous:

Operations do not block the execution.

The next operation can start before the previous one completes.

Example:

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

fs.readFile('/file.md', (err, data) => { // Non-blocking
  if (err) throw err;
  console.log(data);
});
console.log('This message is logged before reading the file completes');
```

Body Parser

Body-parser is a middleware to parse incoming request bodies before your handlers, available under the `req.body` property.

```js
const bodyParser = require('body-parser');
const express = require('express');
const app = express();
app.use(bodyParser.json()); // Parses JSON payload
app.use(bodyParser.urlencoded({ extended: true })); // Parses URL-encoded payload
app.post('/', (req, res) => {
  res.send(req.body);
});
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Summary

Middleware Lifecycle:

Request received.

Middleware functions executed.

Call `next()` to proceed to the next middleware.

Route handler execution.

Types of Middleware:

Application-level. Router-level. Error-handling. Built-in. Third-party.

Application-level Middleware: Bound to the app instance.

Router-level Middleware: Bound to the router instance.

Error-handling Middleware: Handles errors, defined with four arguments.

Third-party Middleware: Community-created, available via npm.

Request Travel in Express:

Client sends request. Passes through middleware stack.

Reaches route handler. Response sent back to client.

Blocking vs Non-blocking Code:

Blocking: Synchronous, sequential execution.

Non-blocking: Asynchronous, concurrent execution.

Body Parser:

Middleware to parse incoming request bodies.

Available under `req.body`.