Codeboosters Tech - Node JS Examples

CODEBOOSTERS TECH

THINK LEARN GRAB

1. 🍔 Food Order Tracker (Timers + Events)

Story:
Imagine you are ordering a burger online. You get notified when the food is delivered after a few minutes. Let's simulate that delivery tracking!

Line-by-Line Explanation:

javascript

```javascript
const EventEmitter = require('events');
```

- **require('events')**: Imports the `events` module from Node.js.
- **EventEmitter**: A class used to create and handle custom events.

javascript

```javascript
const emitter = new EventEmitter();
```

- Creates an instance of `EventEmitter` to use it for emitting (triggering) and listening to events.

javascript

```javascript
function orderFood(food) {
  console.log(`Ordering ${food}...`);
```

- **orderFood function**: Accepts a `food` parameter and logs that the order has been placed.

javascript

```javascript
  setTimeout(() => {
    emitter.emit('delivered', food);
  }, 3000);
}
```

- **setTimeout**: Waits for 3000 milliseconds (3 seconds) and then emits a `delivered` event carrying the food name.

javascript

```javascript
emitter.on('delivered', (food) => {
  console.log(`${food} delivered! Enjoy your meal.`);
});
```

- **emitter.on('delivered', callback)**: Listens for the `delivered` event. When fired, logs that the food has been delivered.

javascript

```javascript
orderFood('Burger');
```

- Calls the function to start the process for a "Burger".

How to approach:

1. **Recognize the event** you want to track (here, "delivered").
2. **Use EventEmitter** to create and listen to that event.

3. **Simulate waiting time** with `setTimeout`.

Final Code:

```javascript
const EventEmitter = require('events');
const emitter = new EventEmitter();

function orderFood(food) {
  console.log(`Ordering ${food}...`);

  setTimeout(() => {
    emitter.emit('delivered', food);
  }, 3000);
}

emitter.on('delivered', (food) => {
  console.log(`${food} delivered! Enjoy your meal.`);
});

orderFood('Burger');
```

---

2. 🎬 Movie Review App (HTTP Server)

Story:
You're building a mini app where users can check movie reviews. If someone visits `/review`, they should see a movie name and rating. Otherwise, they should see a "Page Not Found" error.

Line-by-Line Explanation:

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

- **require('http')**: Loads Node.js's built-in **HTTP module**, used for creating web servers.

```javascript
const server = http.createServer((req, res) => {
```

- **http.createServer(callback)**: Creates an HTTP server.
- **req**: Represents the **request** from the client (browser).
- **res**: Represents the **response** we send back to the client.

```javascript
  if (req.url === '/review') {
```

- Checks if the requested URL is `/review`.
- **req.url**: Gives the path part of the request (like `/review`, `/home`, etc.).

```javascript
    res.writeHead(200, {'Content-Type': 'application/json'});
```

- **res.writeHead(statusCode, headers)**: Sets the HTTP response header.
- `200` means "OK", and we set the content type to **JSON** because we are sending data.

```javascript
    res.end(JSON.stringify({ movie: 'Interstellar', rating: 9 }));
```

- **res.end(data)**: Ends the response by sending a JSON object (converted from JavaScript object using `JSON.stringify()`).

```javascript
  } else {
    res.writeHead(404);
    res.end('Page Not Found');
  }
```

- If the URL is anything else (not `/review`), respond with 404 (Not Found) and a simple text message.

```javascript
});

server.listen(4000, () => {
  console.log('Movie review server at http://localhost:4000');
});
```

- **server.listen(port, callback)**: Starts the server on port 4000 and logs a message when ready.

Important Keywords:

Keyword	Meaning
`http.createServer`	Creates a basic server.
`req`	The incoming client request.
`res`	The response object we send back to the client.
`writeHead`	To set status code and headers.

Keyword	Meaning
`JSON.stringify()`	Converts a JS object into a JSON string for sending.

How to approach:

1. **Decide the endpoint** (`/review`) and the data you want to serve.
2. **Check the URL** when a request comes in.
3. **Send the correct response** based on the URL.
4. **Start the server** on a port (e.g., 4000).

Final Code:

javascript

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

const server = http.createServer((req, res) => {
  if (req.url === '/review') {
    res.writeHead(200, {'Content-Type': 'application/json'});
    res.end(JSON.stringify({ movie: 'Interstellar', rating: 9 }));
  } else {
    res.writeHead(404);
    res.end('Page Not Found');
  }
});

server.listen(4000, () => {
  console.log('Movie review server at http://localhost:4000');
});
```

3. 🍩 Bakery Order System (Callbacks)

Story:
Imagine you order a cake from a bakery. It takes some time to bake, and once it's ready, you get notified. We'll simulate baking a cake and alerting when it's done using **callbacks**!

Line-by-Line Explanation:

```javascript
function bakeCake(flavor, callback) {
```

- **bakeCake function**: Accepts two parameters:
  - `flavor`: Type of cake (like chocolate, vanilla).
  - `callback`: A function that will run when baking is complete.

```javascript
  console.log(`Baking ${flavor} cake...`);
```

- Logs that the cake baking process has started.

```javascript
  setTimeout(callback, 2000);
```

- **setTimeout(callback, time)**: Waits for 2000 milliseconds (2 seconds), then executes the `callback` function.

- Simulates baking time.

```javascript
}
```

- End of `bakeCake` function.

```javascript
bakeCake('chocolate', () => {
  console.log('Cake ready! 🎂');
});
```

- Calls the `bakeCake` function:
  - Passes `'chocolate'` as the flavor.
  - Passes a callback function that logs `'Cake ready! 🎂'` when the cake is done baking.

Important Keywords:

Keyword	Meaning
`callback`	A function passed into another function to run later.
`setTimeout`	Delays the execution of code after a certain time.

How to approach:

1. **Understand the task:** Do something after a delay (baking -> after 2 sec -> ready).
2. **Use a callback function** that will execute once baking is done.
3. **Use** `setTimeout` to simulate delay.

Final Code:

```javascript
function bakeCake(flavor, callback) {
  console.log(`Baking ${flavor} cake...`);
  setTimeout(callback, 2000);
}

bakeCake('chocolate', () => {
  console.log('Cake ready! 🎂');
});
```

4. 📦 Package Delivery Tracker (Events)

Story:
Think about an online shopping site. After you order a product, you get a message when it’s shipped. Let’s simulate package shipping updates using **events**!

Line-by-Line Explanation:

```javascript
const EventEmitter = require('events');
```

- **require('events'):** Loads the built-in `events` module from Node.js.
- **EventEmitter**: A special class to **create custom events** and **respond to them**.

```javascript
const delivery = new EventEmitter();
```

- Creates a new **EventEmitter instance** named `delivery` to manage package-related events.

```javascript
delivery.on('shipped', (packageId) => {
  console.log(`Package ${packageId} shipped!`);
});
```

- **delivery.on('shipped', callback):**
  - Listens for the `'shipped'` event.
  - When `'shipped'` is emitted, this function will run.
  - It logs the message along with the `packageId`.

```javascript
delivery.emit('shipped', 1234);
```

- **delivery.emit('shipped', 1234):**
  - Triggers (fires) the `'shipped'` event immediately.
  - Passes `1234` as the **packageId** to the listener.

Important Keywords:

Keyword	Meaning
`EventEmitter`	Class to manage events in Node.js.

Keyword	Meaning
`on(event, callback)`	Listen for an event and handle it.
`emit(event, data)`	Trigger an event with optional data.

How to approach:

1. **Identify the event** you want to track (`'shipped'` here).
2. **Create an EventEmitter** instance to manage it.
3. **Listen for the event** using `.on()`.
4. **Trigger the event** using `.emit()` when ready.

Final Code:

javascript

```javascript
const EventEmitter = require('events');
const delivery = new EventEmitter();

delivery.on('shipped', (packageId) => {
  console.log(`Package ${packageId} shipped!`);
});

delivery.emit('shipped', 1234);
```

5. 🧺 Laundry Status Update (Timers)

Story:
Imagine putting your laundry into the washing machine. It takes time to complete, and once done, the machine notifies you. Let's simulate the wash cycle using a simple **timer**!

Line-by-Line Explanation:

javascript

```javascript
console.log("Washing started...");
```

- **console.log()**: Prints `"Washing started..."` to show that the laundry process has begun.

javascript

```javascript
setTimeout(() => {
  console.log("Washing completed!");
}, 5000);
```

- **setTimeout(callback, delay)**:
  - Waits for 5000 milliseconds (5 seconds).
  - Then runs the callback function, which prints `"Washing completed!"`.
- This simulates the washing time — after 5 seconds, you get a message that it's done.

Important Keywords:

Keyword	Meaning
`console.log()`	Used to print messages to the console.

Keyword	Meaning
`setTimeout()`	Runs a function after a specific time delay.

How to approach:

1. **Start the task** by logging a message.
2. **Use a timer** (`setTimeout`) to delay the next action.
3. **After the delay**, run the callback to say the task is complete.

Final Code:

```javascript
console.log("Washing started...");
setTimeout(() => {
  console.log("Washing completed!");
}, 5000);
```

CODEBOOSTERS TECH
THINK LEARN GRAB

Codeboosters Tech
team_codeboosters
www.codeboosters.in

6. 🚕 Taxi Booking Service (Events + Timers)

Story:
Imagine you book a taxi online. You get a message when the taxi arrives at your location after a few minutes. Let's simulate this using **events** and **timers**!

Line-by-Line Explanation:

```javascript
const EventEmitter = require('events');
```

- **require('events')**: Loads the `events` module, needed to create and manage events in Node.js.

```javascript
const taxi = new EventEmitter();
```

- Creates a new **EventEmitter instance** called `taxi` to manage taxi-related events.

```javascript
taxi.on('arrived', (location) => {
  console.log(`Taxi arrived at ${location}`);
});
```

- **taxi.on('arrived', callback):**
  - Listens for an `'arrived'` event.
  - When fired, it logs that the taxi has arrived at the given `location`.

```javascript
function bookTaxi(location) {
  console.log(`Booking taxi to ${location}...`);
```

- **bookTaxi function:**

- Takes the `location` as input.
- Logs that a taxi is being booked to that destination.

```javascript
  setTimeout(() => taxi.emit('arrived', location), 3000);
}
```

- **setTimeout(callback, 3000):**
  - Waits for 3 seconds (3000 milliseconds).
  - Then **emits** the `'arrived'` event with the location.

```javascript
bookTaxi('Airport');
```

- Calls the `bookTaxi` function with `'Airport'` as the destination.
- After 3 seconds, it will print that the taxi has arrived at the Airport.

Important Keywords:

Keyword	Meaning
`EventEmitter`	Creates and manages events.
`on(event, callback)`	Set up an event listener.
`emit(event, data)`	Trigger an event.
`setTimeout()`	Waits before executing code.

How to approach:

1. **Create an event** that represents the taxi arriving.

2. **Listen** for the `'arrived'` event using `.on()`.
3. **Use a timer** (`setTimeout`) to simulate travel time.
4. **Emit** the event once the taxi reaches the location.

Final Code:

```javascript
const EventEmitter = require('events');
const taxi = new EventEmitter();

taxi.on('arrived', (location) => {
  console.log(`Taxi arrived at ${location}`);
});

function bookTaxi(location) {
  console.log(`Booking taxi to ${location}...`);
  setTimeout(() => taxi.emit('arrived', location), 3000);
}

bookTaxi('Airport');
```

---

7. 🎨 Art Gallery API (Simple HTTP API)

Story:
Imagine you are building an art gallery API where users can get a list of famous artworks. If

they visit the `/artworks` endpoint, they should see a list of paintings. If they request something else, they get a "Not Found" error.

Line-by-Line Explanation:

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

- **require('http')**: Loads Node.js's **HTTP module**, which is essential for creating an HTTP server to handle web requests.

```javascript
const artworks = ['Mona Lisa', 'Starry Night', 'The Scream'];
```

- **artworks array**: An array of strings that represent the names of famous artworks that will be returned by the API.

```javascript
const server = http.createServer((req, res) => {
```

- **http.createServer(callback)**: Creates a server to handle incoming requests.
  - **req**: Represents the incoming request from the client (browser).
  - **res**: Represents the server's response back to the client.

```javascript
  if (req.url === '/artworks') {
```

- **req.url**: Checks the URL of the incoming request. If it's `/artworks`, the server will respond with the list of artworks.

```javascript
```

```javascript
    res.writeHead(200, {'Content-Type': 'application/json'});
```

- **res.writeHead(200, {...})**: Sends a response with status code 200 (OK) and sets the **Content-Type** to `application/json`. This tells the browser to expect a JSON response.

```javascript
    res.end(JSON.stringify(artworks));
```

- **res.end(data)**: Ends the response by sending the **artworks array** as a JSON string.
  - **JSON.stringify(artworks)** converts the array into a JSON-formatted string.

```javascript
  } else {
    res.writeHead(404);
    res.end('Artwork not found');
  }
```

- If the URL is **not** `/artworks`, the server responds with a **404 Not Found** error and a message `"Artwork not found"`.

```javascript
});

server.listen(5000, () => {
  console.log('Art gallery API at http://localhost:5000');
});
```

- **server.listen(5000, callback)**: Starts the server on port 5000. Once it’s running, it logs a message showing the URL (`http://localhost:5000`) where the API can be accessed.

Important Keywords:

Keyword	Meaning
`http.createServer`	Creates an HTTP server to handle requests.
`req.url`	The requested URL (path) from the client.
`res.writeHead()`	Sets the HTTP status code and response headers.
`JSON.stringify()`	Converts a JavaScript object or array into a JSON string.
`server.listen()`	Starts the server on a specific port and listens for incoming requests.

How to approach:

1. **Set up a simple server** to handle HTTP requests.
2. **Check the URL** in the request to determine what data to return (e.g., `/artworks`).
3. **Respond with JSON** data using `JSON.stringify()`.
4. **Handle errors** by returning a 404 status for unknown URLs.

Final Code:

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

const artworks = ['Mona Lisa', 'Starry Night', 'The Scream'];

const server = http.createServer((req, res) => {
  if (req.url === '/artworks') {
    res.writeHead(200, {'Content-Type': 'application/json'});
    res.end(JSON.stringify(artworks));
  } else {
    res.writeHead(404);
    res.end('Artwork not found');
  }
});
```

```
server.listen(5000, () => {
  console.log('Art gallery API at http://localhost:5000');
});
```

8. 🥤 Smoothie Maker (Callback + Timer)

Story:
Imagine you're making a smoothie at home. It takes a few seconds to prepare, and once it's ready, you get notified. Let's simulate the smoothie-making process using **callbacks** and **timers**!

Line-by-Line Explanation:

```javascript
function makeSmoothie(flavor, callback) {
```

- **makeSmoothie function**: Accepts two parameters:
  - `flavor`: The flavor of the smoothie (e.g., Mango, Strawberry).
  - `callback`: A function that will be executed once the smoothie is ready.

```javascript
```

```
  console.log(`Making ${flavor} smoothie...`);
```

- **console.log()**: Logs that the smoothie-making process has started, showing the flavor being prepared.

```javascript
  setTimeout(() => {
    console.log(`${flavor} smoothie ready! 🥤`);
    callback();
  }, 2500);
```

- **setTimeout(callback, 2500)**:
  - Waits for **2500 milliseconds (2.5 seconds)** before running the `callback` function.
  - After the timer ends, it logs that the smoothie is ready and calls the `callback` to notify the user.

```javascript
}
```

- End of the `makeSmoothie` function.

```javascript
makeSmoothie('Mango', () => {
  console.log('Enjoy your drink!');
});
```

- Calls the `makeSmoothie` function with `'Mango'` as the flavor.
- Once the smoothie is ready (after 2.5 seconds), the callback logs `'Enjoy your drink!'`.

Important Keywords:

Keyword	Meaning
`callback`	A function passed into another function to be executed later.
`setTimeout()`	Delays the execution of a function by a specified time (in milliseconds).

How to approach:

1. **Start the task** (making the smoothie) and log it.
2. **Use** `setTimeout` to simulate the waiting time for smoothie preparation.
3. **Call the callback function** once the smoothie is ready to notify the user.

Final Code:

```javascript
function makeSmoothie(flavor, callback) {
  console.log(`Making ${flavor} smoothie...`);
  setTimeout(() => {
    console.log(`${flavor} smoothie ready! 🥤`);
    callback();
  }, 2500);
}

makeSmoothie('Mango', () => {
  console.log('Enjoy your drink!');
});
```

CODEBOOSTERS TECH
THINK LEARN GRAB

Codeboosters Tech
team_codeboosters
www.codeboosters.in

9. 🛍️ Shopping Cart Server (HTTP POST simulation)

Story:
Imagine you're shopping online. When you add an item to your cart, the server receives that item and acknowledges it. Let’s simulate the shopping cart feature using an HTTP **POST** request!

Line-by-Line Explanation:

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

- **require('http')**: Loads the built-in Node.js **HTTP module**, which allows us to create a server and handle HTTP requests.

```javascript
const server = http.createServer((req, res) => {
```

- **http.createServer(callback)**: Creates an HTTP server that listens for requests.
  - **req**: Represents the incoming request.
  - **res**: Represents the server’s response.

```javascript
  if (req.method === 'POST' && req.url === '/cart') {
```

- **req.method**: Checks the method of the incoming request. In this case, we're looking for a **POST** request.
- **req.url**: Checks the URL of the request. If the request is made to `/cart`, we proceed with handling the cart item.

```javascript
  let body = '';
  req.on('data', chunk => {
    body += chunk.toString();
  });
```

- **let body = ''**: Initializes an empty string to accumulate the data from the incoming request.
- **req.on('data', callback)**: The `data` event is triggered as the request body comes in chunks.
  - Each chunk is added to the `body` string.
  - `chunk.toString()` converts the chunk from a buffer to a string.

```javascript
  req.on('end', () => {
    console.log(`Cart item added: ${body}`);
    res.end('Item added to cart');
  });
```

- **req.on('end', callback)**: The `end` event triggers once all the data has been received.
  - Logs the item added to the cart (`body` contains the cart item).
  - **res.end()**: Ends the response and sends a message back to the client that the item has been added to the cart.

```javascript
} else {
  res.statusCode = 404;
  res.end('Not Found');
}
```

- If the request method isn't **POST** or the URL isn't `/cart`, it responds with a **404 Not Found** error and a message.

```javascript
});

server.listen(6000, () => {
  console.log('Shopping cart server running at http://localhost:6000');
});
```

- **server.listen(6000, callback):** Starts the server on port 6000. Once it's running, it logs that the shopping cart server is live at `http://localhost:6000`.

Important Keywords:

Keyword	Meaning
`http.createServer()`	Creates an HTTP server to handle requests.
`req.method`	The HTTP method of the incoming request (e.g., GET, POST).
`req.on('data')`	Event triggered when chunks of data are received from the client.
`req.on('end')`	Event triggered once all the data is received.
`res.end()`	Sends the response to the client and ends the request-response cycle.

How to approach:

1. **Set up a POST endpoint** (`/cart`) to receive data.
2. **Listen for incoming data** chunks and build the body.
3. **Once data is complete**, log the item and send a response to the client.
4. **Handle errors** by responding with a 404 for unsupported methods or URLs.

Final Code:

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

const server = http.createServer((req, res) => {
  if (req.method === 'POST' && req.url === '/cart') {
    let body = '';
    req.on('data', chunk => {
      body += chunk.toString();
    });

    req.on('end', () => {
      console.log(`Cart item added: ${body}`);
      res.end('Item added to cart');
    });
  } else {
    res.statusCode = 404;
    res.end('Not Found');
  }
});

server.listen(6000, () => {
  console.log('Shopping cart server running at http://localhost:6000');
});
```

---

10. 🍿 Popcorn Ready Timer (Timers)

Story:
Imagine you start the popcorn machine, and after a few seconds, it notifies you that your popcorn is ready. Let's simulate this using a **timer**!

Line-by-Line Explanation:

```javascript
console.log("Starting popcorn machine...");
```

- **console.log()**: Prints the message `"Starting popcorn machine..."` to the console to indicate that the popcorn machine has started.

```javascript
setTimeout(() => {
  console.log("Popcorn ready! 🍿");
}, 4000);
```

- **setTimeout(callback, 4000)**:
  - Waits for **4000 milliseconds (4 seconds)** before executing the callback.
  - Once the 4 seconds are over, it logs `"Popcorn ready! 🍿"` to the console, simulating that the popcorn is ready.

Important Keywords:

Keyword	Meaning
`console.log()`	Logs messages to the console.
`setTimeout()`	Delays the execution of a function by a specified time.

How to approach:

1. **Start the task** by logging a message about the popcorn machine.
2. **Use** `setTimeout()` to simulate the time it takes for the popcorn to be ready.
3. **Log a message** when the task is completed (after the delay).

Final Code:

```javascript
console.log("Starting popcorn machine...");
setTimeout(() => {
  console.log("Popcorn ready! 🍿");
}, 4000);
```