Introduction to Fullstack Development

IIT Kharagpur

January 29, 2026

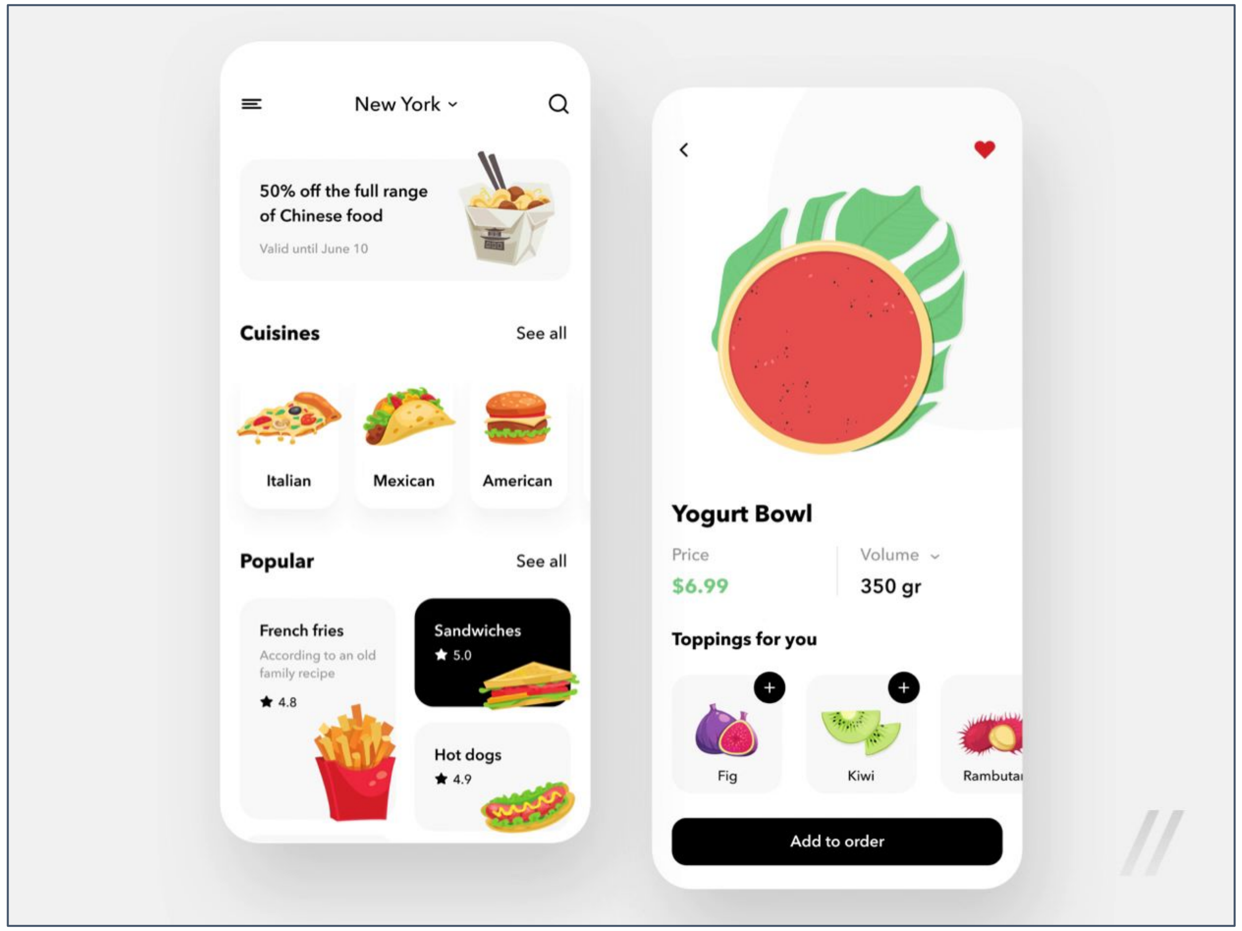
Fullstack in Action: The Food Delivery App

Frontend: What You See & Do

This is the interactive part users experience directly on their device.

User Interactions:

- Browse restaurants and their menus.
- Select food items and add them to a cart.
- Enter delivery details and payment information.
- Confirm the order and view its real-time status.



- The frontend is built with technologies like **React**, **Angular**, or **Vue.js** for web, or **Swift/Kotlin** for mobile apps.

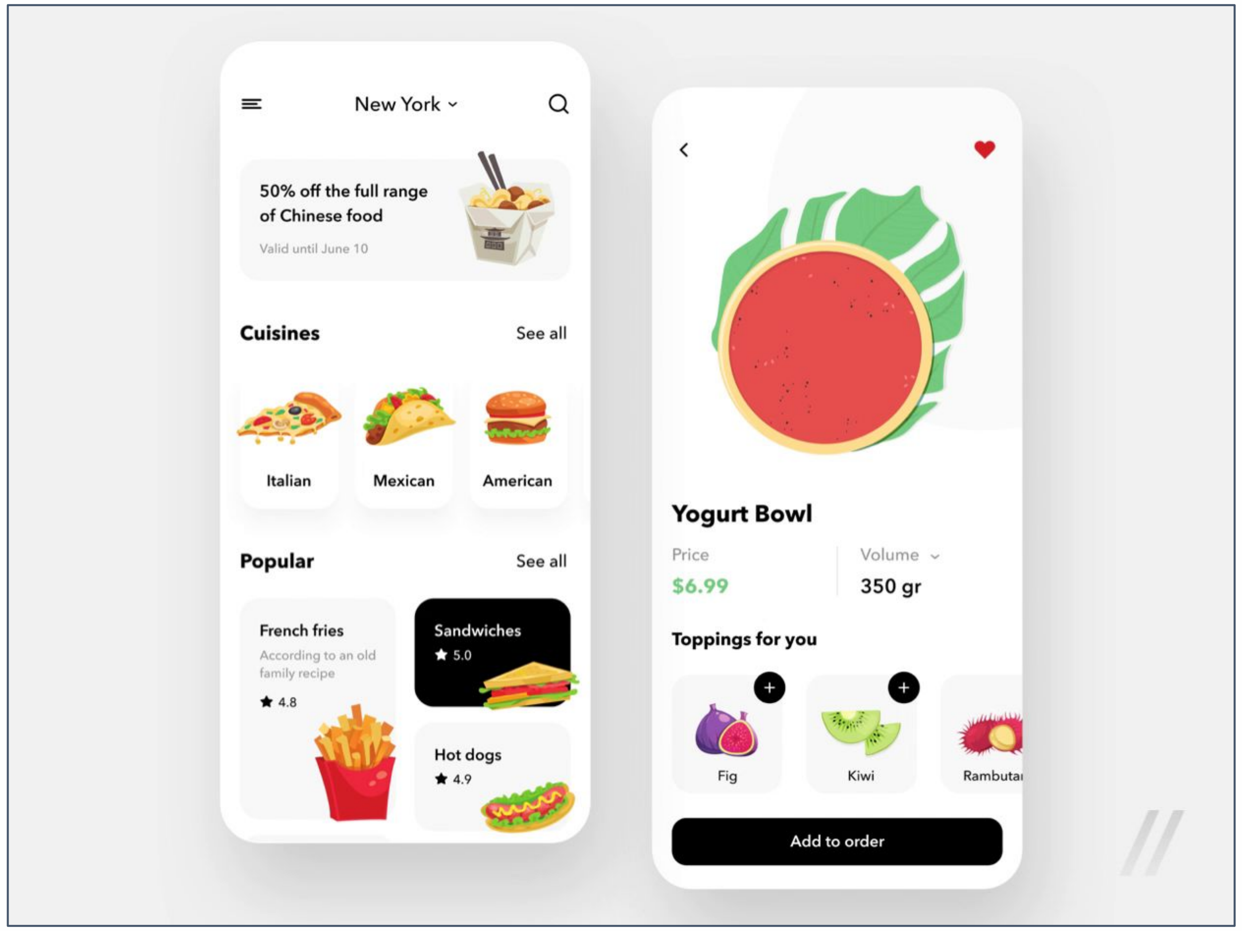
Fullstack in Action: The Food Delivery App

Backend: What Happens Behind the Scenes

The backend handles all the logic, data storage, and server operations.

Server-Side Processes:

- Authenticates user logins and manages profiles.
- Retrieves restaurant data, menus, and pricing.
- Validates orders, processes payments, and updates inventory.
- Coordinates with restaurants and delivery drivers.
- Stores all critical information in a database (e.g., PostgreSQL, MongoDB).



- Common backend technologies include **Node.js**, **Python/Django**, **Ruby on Rails**, and **Java/Spring Boot**.

What Happens When a User Clicks a Button?

1. **User Clicks**
The user interacts with the UI, eg: clicking “Add to Cart” or “Place Order”.

2. **Function Mapped**
The click triggers a specific function.

3. **HTTP Request**
The function sends a request to the backend.

4. **Backend Processes**
The server handles the request, eg: checking if inventory has the item, fetching the menu.

5. **Backend Responds**
The server sends data back, eg: status of transaction, sending back the menu.

6. **UI Updates**
The user interface reflects new data, eg: item added, order confirmed.

This Flow Exists Everywhere

Frontend

Click events in the user interface.

Backend

Request events handled by servers.

Operating System

Input/output (IO) completion events.

Observation

Every system reacts to “something happening.” This fundamental principle is the very essence of **event-driven architecture**.

How Early Web Applications Worked

Early Model: One Request, One Thread

In the early web development:

- Each incoming request was typically assigned its own dedicated thread.
- **One request** → **one thread**.
- The thread **blocks** while performing IO operations.
- This includes reading files, querying databases, and waiting for network responses.

Consequences of Blocking

This blocking behaviour led to inefficiencies as demands grew.

- Many threads consume substantial system resources.
- **High memory usage** due to numerous concurrent threads.
- **Poor scalability**, as resource contention limits the number of users served.

The Core Problem With Blocking Servers

Thread Waits Idle

A blocking thread does nothing, consuming resources.

Cannot Serve Others

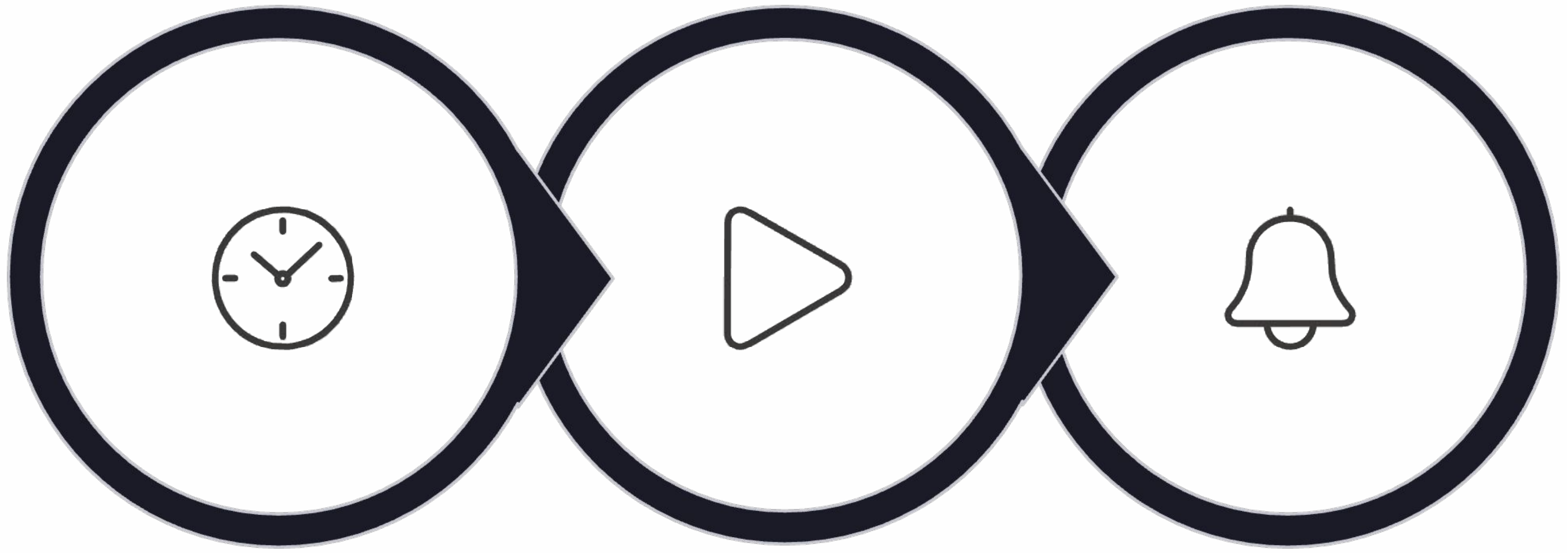
While blocked, the thread is unavailable for new requests.

Wasted Hardware

CPU and memory are underutilised while waiting for IO.

Consider a scenario: 1,000 active users could mean 1,000 threads, with a **majority often sitting idle**. This highlights a critical limitation where computational **resources are inefficiently used**.

Why Event-Driven Architecture Was Introduced



Avoid Waiting

Start IO, Move On

React When Finished

The primary objective of event-driven architecture is to **eliminate the inefficiencies** associated with blocking operations. **Instead of waiting** for a task to complete, the system initiates the task, moves on to other work, and only **reacts** when the initial task **signals its completion**.

Event-Driven Model: Correct and Simple

The Core Model

- **Register interest** in specific events.
- **Continue executing** other code without delay.
- **Handle** the **event** only when it occurs.

Important Distinctions

- It is **not parallel execution**; tasks run sequentially.
- There is **no idle waiting** for operations to finish.
- Typically processes one event to completion at a time.

This model ensures efficient resource utilisation by constantly doing useful work rather than pausing for slow operations.

Backend View – What Is an Event?

HTTP Request

An incoming web request from a client.

File Read

Completion of data retrieval from disk.

Database Query

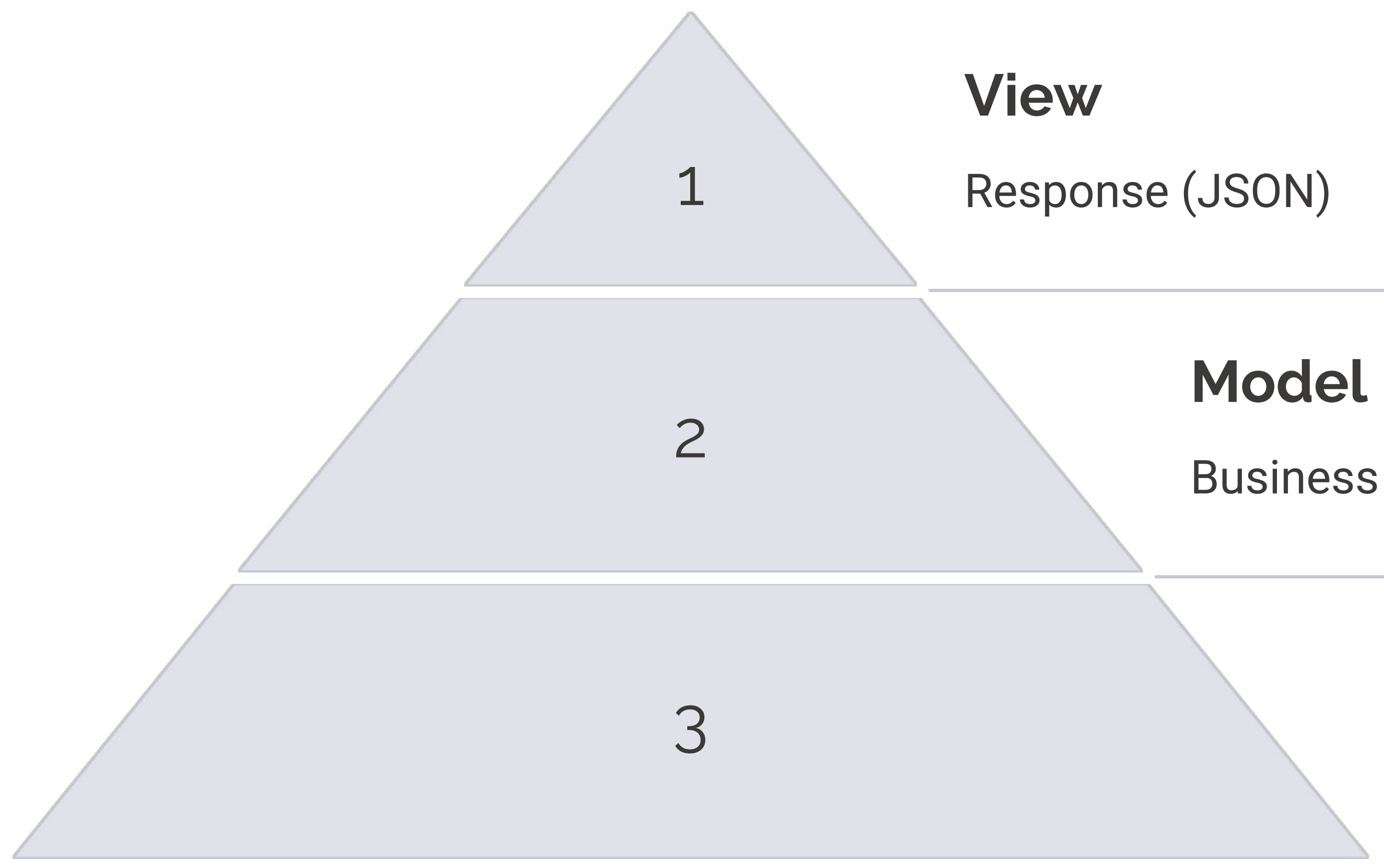
The result of a data retrieval or modification operation.

Timer Expiry

A scheduled task or delay reaching its end.

Each of these events carries specific data relevant to its occurrence and is associated with a pre-defined handler function that dictates how the system should respond.

Backend Architecture Before Code (MVC)



View

Response (JSON)

Model

Business Logic + Data

Controller

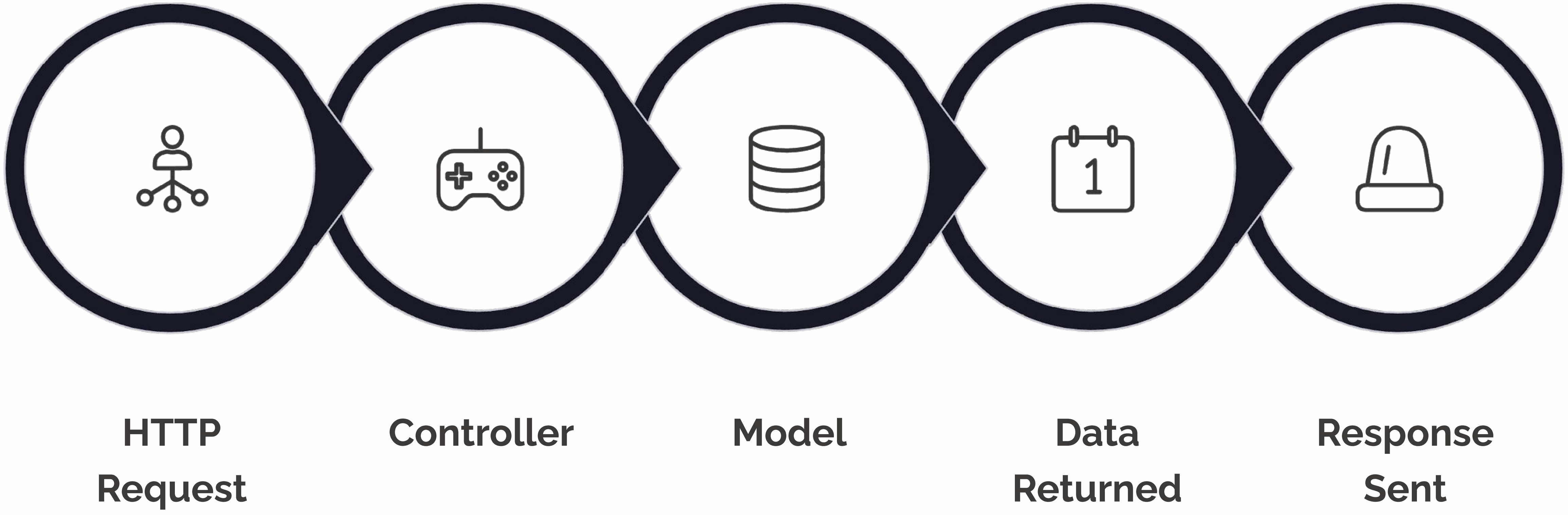
Handles Event (Request)

Why Structure Matters

Event-driven systems can **scale** rapidly, but without a clear architectural pattern, their codebase can quickly become **unmanageable**.

The Model-View-Controller (MVC) pattern provides a robust framework to prevent this.

MVC Request Flow



Benefit: Clear Responsibility per Layer

This structured approach ensures that each component of the application has a distinct and well-defined role, making the system easier to develop, debug, and maintain. The separation of concerns within MVC is crucial for complex backend systems.

Why Node.js Fits This Model

1

Single JavaScript Thread

Node.js operates on a single execution thread, simplifying concurrency models.

2

Non-Blocking I/O

It excels at handling input/output operations without pausing execution, crucial for responsiveness.

3

Event-Driven Execution

Its core design is built around reacting to events as they occur, aligning perfectly with this architecture.

What Node Avoids

Node.js deliberately **avoids** the traditional **thread-per-request model**, enabling it to manage a high volume of concurrent connections efficiently without the overhead of multiple threads.

What is a Callback?

Definition

- A **callback function** is a function passed as an argument to another function.
- This “callback” gets executed inside the outer function
- This allows the code to be run after a specific operation or event completes.

Why They Matter

- Callbacks are crucial for managing **asynchronous operations**
- It can help in things like fetching data from a server or setting a timer
- They ensure that certain code executes only when a prior, time-consuming task has finished, preventing the program from blocking and waiting.

Code Example (JavaScript)

```
function executeAfterDelay(callback) {
  setTimeout(() => {
    console.log("Operation complete!");
    callback(); // Call the passed function
  }, 1000);
}

function doSomethingElse() {
  console.log("Now doing something else.");
}

executeAfterDelay(doSomethingElse);
// Output after 1 second:
// Operation complete!
// Now doing something else.
```

Another Callback Example: Backend Operation

```
readFile("data.txt", (err, data) => { // runs after file is ready });
```

Let's break down this common backend scenario involving asynchronous file reading.

1

File Read Initiated

The `readFile` function begins the process of reading `data.txt`.

2

JS Continues Execution

While the file is being read, the JavaScript thread is free to handle other tasks.

3

Callback Queued

Once `data.txt` is fully read, the associated callback function is placed in the event queue.

Node's Event Loop: A Mental Model

1

Take Next Event Callback

The event loop continuously checks for pending tasks in the event queue.

2

Run It Fully

Once a callback is picked, it executes to completion without interruption.

3

Return to Loop

After execution, the loop becomes available to process the next event.

Key Rule:

No callback is interrupted once it starts, ensuring predictable JavaScript execution flow.

Event Queue: What Waits Here

- **Completed HTTP Requests**

  Callbacks associated with resolved network requests are placed here.

- **Completed File Reads**

  Functions triggered upon successful completion of file system operations.

- **Completed DB Queries**

  Callbacks for database operations that have finished their work.

> Crucial Point:
>
> Only **ready** work tasks whose dependencies are met, are added to the event queue, ensuring efficiency.

Express.js: Turning Events into Routes

Express.js simplifies backend development by mapping incoming HTTP events to specific functions.

URL to Function Mapping

Connects specific URLs to corresponding handler functions.

Request Parsing

Extracts data from incoming HTTP requests (e.g., body, query parameters).

Response Helpers

Provides utilities for constructing and sending HTTP responses.

For example: `app.get("/data", handler);`

Event:

An incoming HTTP request to `/data`.

Handler:

The designated controller function to process that request.

Frontend: Facing Similar Challenges

Before modern frameworks, frontend development grappled with significant issues.

Direct DOM Manipulation

Manually updating the Document Object Model led to spaghetti code and maintainability nightmares.

Hard to Track State Changes

Keeping track of how user interactions altered the application's state was complex and error-prone.

Tight Coupling

UI components were often heavily dependent on each other, making changes difficult and risky.

This often resulted in an increase in bugs, an inconsistent user interface, and overly complex codebases.

React's Core Idea: UI as a Function of State

React revolutionised frontend development by providing elegant solutions to these challenges.

How UI Reacts to Events

React offers a declarative way to describe UI behaviour in response to user actions.

How UI Updates Consistently

It ensures that your user interface always reflects the current state of your application reliably.

The Fundamental Principle:

UI = function(state)

Your UI is simply a pure function of your application's state, making it predictable and testable.

React Component: Event Handler + UI

In React, a component encapsulates both its visual representation and its interactive behaviour.

```
function Button() { return Click; }
```

UI Description

Defines what the component looks like (e.g., a button with specific text).

Event Handlers

Functions that respond to user interactions, such as clicks or input changes.

Local State

Internal data that influences the component's rendering and behaviour.

Event Handling in React

React's approach to event handling mirrors the asynchronous nature seen in backend systems.

User Click

A user interaction, like clicking a button, triggers an event.

Function Execution

This event directly leads to the execution of a defined handler function.

This model ensures a **consistent** and **understandable** way to manage interactivity, much like how backend events are processed.

State Change Drives UI Update

The true power of React lies in its efficient and automatic UI updates based on state changes.

```
setCount(count + 1);
```

Detects State Change

React monitors component state for any modifications.

Re-renders Component

Upon detecting a change, React intelligently re-evaluates the component.

Updates DOM Efficiently

It then updates only the necessary parts of the actual DOM, optimising performance.

Side Effects: Managing External Interactions

The `useEffect` hook allows React components to interact with the outside world.

```
useEffect(() => { fetch("/data"); }, []);
```

Network Calls

Fetching data from APIs or external services.

Timers

Setting up delays, intervals, or timeouts.

Subscriptions

Subscribing to external data sources and cleaning up afterwards.

Side effects are operations that affect something outside the component's direct render logic, ensuring clean separation of concerns.

Styling as a Reaction: The Tailwind Approach

Data-Driven UI Styling

- Tailwind CSS champions a paradigm where UI styles are directly driven by data attributes, moving away from traditional imperative CSS.

No Imperative CSS Logic

- This approach eliminates the need for complex, imperative CSS logic, simplifying development and maintenance.

Utility-First Classes

- Styles are applied using utility classes directly in the HTML, such as `className="bg-blue-500 px-4"`, for rapid prototyping and consistent design.

Putting It All Together: The Runtime Story

An End-to-End Journey

- **User Click:** The interaction begins with a user's click.
- **React Handler:** A React handler captures the event.
- **HTTP Request:** An HTTP request is dispatched to the backend.
- **Express Controller:** An Express controller processes the request.
- **Model Logic:** Business logic is executed via the model.
- **JSON Response:** A JSON response is sent back.
- **React State Update:** React state updates based on the response.
- **UI Re-render:** The user interface re-renders to reflect changes.

This sequential flow ensures a responsive and dynamic user experience in modern web applications.

Why This Architecture Works

Key Benefits of Event-Driven Design

- **No Idle Waiting:** Asynchronous operations minimise waiting times, improving overall system responsiveness and user experience.
- **Clear Separation of Concerns:** Each component has a distinct responsibility, making the system easier to understand, develop, and maintain.
- **Scales with Users:** The architecture is inherently scalable, efficiently handling an increasing number of concurrent users and requests.
- **Same Mental Model Everywhere:** A consistent mental model across the stack simplifies development and collaboration among teams.
- Event-driven architecture is a fundamental response to the inherent cost of waiting in distributed systems.
- By embracing an event-driven paradigm, we address bottlenecks and unlock greater efficiency and scalability.

Further Learning Resources

To deepen your understanding and continue your journey in modern web development, explore these essential documentation links:

- Getting started with Node.js - Guide to install Node.js
- Installing Express.js - Guide to install Express.js
- React installation guide - Guide to install React
- Tailwind CSS installation guide - Guide to install Tailwind CSS
- HTML Documentation - Your go-to reference for fundamental web markup
- Tailwind CSS Documentation - Learn more about utility-first CSS for rapid UI development
- ReactJS Documentation - The official guide for building dynamic user interfaces with React
- Node.js Documentation - Explore the server-side JavaScript runtime for scalable backend applications
- Express.js Documentation - Discover the minimal and flexible Node.js web application framework
- Git tutorial - Beginner friendly guide to getting started with Git

Full Demo: https://github.com/meabhisingh/mernProjectEcommerce

Also the YouTube links given in the README of this Repo are good.