Week 5: Asynchronous JavaScript, Promises, and Node.js Events

Day 1: Synchronous vs. Asynchronous Execution

The Single-Threaded Nature of JavaScript

JavaScript is inherently a single-threaded language, meaning it possesses one call stack and can execute only one command at a time. In a synchronous execution model, each line of code must finish before the next one begins. If a function performs a heavy task, such as fetching data for the **Gadget API**, the entire application becomes "blocked" and unresponsive until that task completes.

The Non-Blocking I/O Model

Asynchronous programming allows JavaScript to initiate a long-running task and then move on to the next instruction without waiting. This is achieved through the use of callbacks and promises, allowing for concurrency where multiple tasks appear to happen simultaneously even though they share a single thread.

Day 2: The Event Loop and Callback Functions

The Event Loop Architecture

The Event Loop is the mechanism that allows Node.js to perform non-blocking I/O operations. It consists of several phases, including timers, pending callbacks, and the poll phase. The loop constantly checks if the call stack is empty; if it is, it pushes tasks from the callback queue onto the stack for execution.

Implementing Callbacks

A callback is a function passed as an argument to another function, which is then invoked after an asynchronous operation finishes.

- **Creation**: You define a function that will handle data once it is ready.
- **Execution**: The main function calls this "callback" back once its internal logic is done.
- **Issues**: Over-reliance on nested callbacks leads to "callback hell," making code difficult to read.
- **The 'this' Keyword**: In callback functions, the this keyword often loses its original context, requiring developers to use arrow functions or .bind() to maintain proper object reference.

Day 3: Promises and the Promise Lifecycle

What is a Promise?

Introduced in ECMAScript 2015, a Promise is an object that represents the eventual completion or failure of an asynchronous operation. It provides a cleaner alternative to callbacks by allowing for "chaining".

The Three States of a Promise

1. **Pending**: The initial state; the operation has not started or is still in progress.
2. **Fulfilled**: The operation completed successfully, and the promise now has a resulting value.
3. **Rejected**: The operation failed, and the promise contains an error or reason for the failure.

Promise Chaining

Instead of nesting functions, developers use .then() for successful results and .catch() for error handling. This allows the output of one asynchronous task to be passed directly into the next, maintaining a flat and readable code structure.

Day 4: Async/Await and Error Handling

Syntactic Sugar for Promises

The async and await keywords allow developers to write asynchronous code that looks and behaves like synchronous code. An async function always returns a promise, and the await keyword pauses execution until the promise is settled.

Try-Catch Blocks

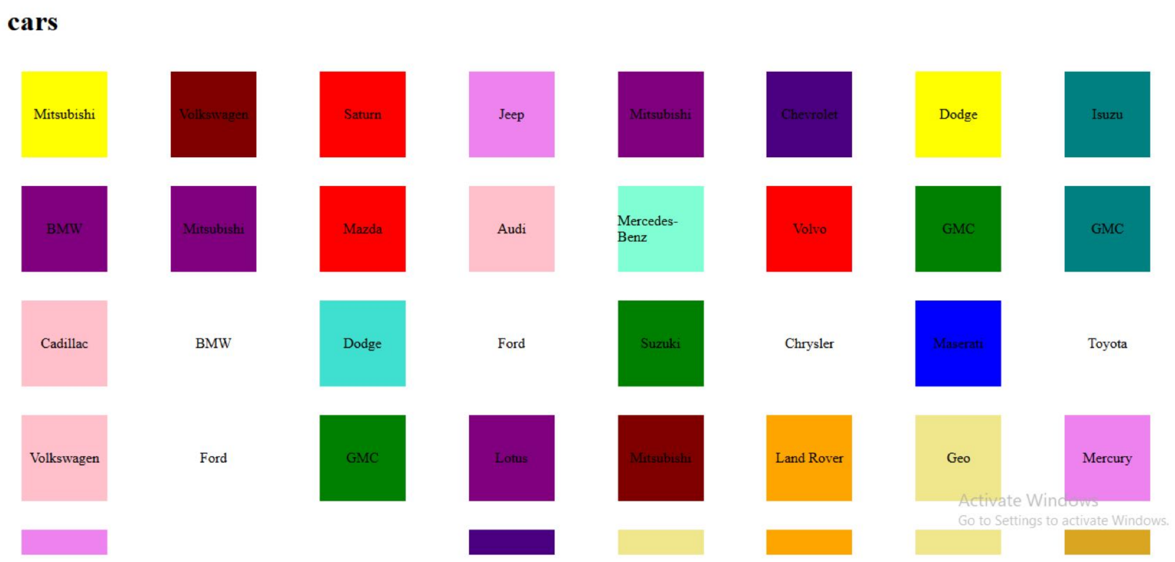
Because async/await uses a linear structure, standard try...catch blocks can be used for error handling. This is more robust than .catch() because it can handle both asynchronous errors and synchronous logic errors in the same block. This is the primary method used in your **GadgetShop** and **LMS** folders to fetch data from the server.

Day 5: Event Driven Programming and Event Emitters

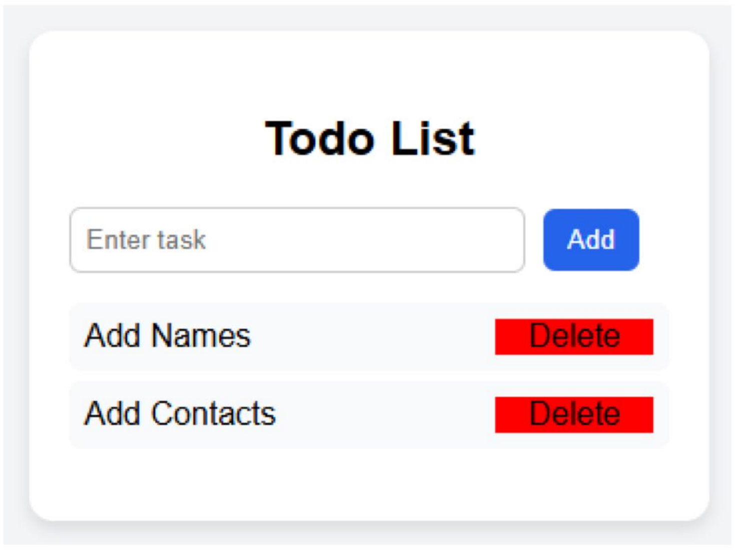
The EventEmitter Class

Node.js core is built on an event-driven architecture where certain objects (emitters) emit named events that cause "listener" functions to be called.

- **Event Loop and Handlers**: The event loop listens for these signals and triggers the associated handler.
- **Inheritance**: Many classes in Node.js, such as the Request and Response objects in Express, inherit from the EventEmitter class to handle data streams and connections.
- **Core Functions**: Developers use .on() to register a listener and .emit() to signal that an event has occurred.
- **Context Handling**: Just like in callbacks, handling the this keyword within event listeners is a common challenge that requires careful function binding.



Task 1 Cars



Task 2 TodoList