Modern JavaScript Essentials

A comprehensive guide to advanced JavaScript concepts for modern web development. This presentation covers crucial aspects of modern JavaScript, from language fundamentals to asynchronous patterns and React integration.

Whether you're refining your skills or starting your JavaScript journey, these concepts form the foundation of contemporary web application development.

The `this` Keyword in JavaScript

Context Rules

The `this` keyword is one of JavaScript's most powerful yet confusing features. It dynamically refers to the context of function invocation, not where the function was defined.

In strict mode, standalone functions have `this` as `undefined` rather than the global object, preventing accidental global variable creation.

Method Context

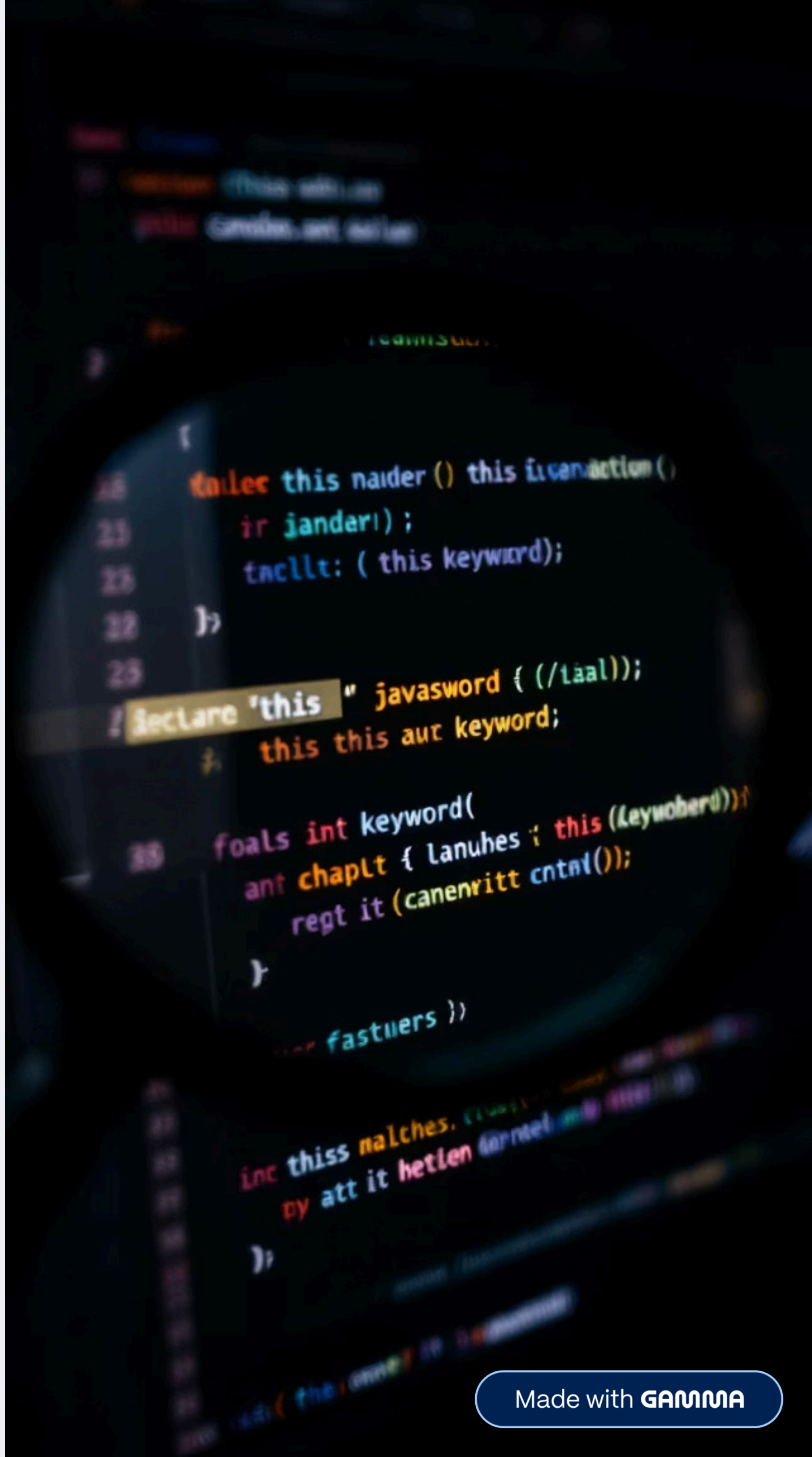
When used in an object method, `this` refers to the calling object:

```
const user = {
  name: 'John',
  greet() {
    console.log(`Hello,
${this.name}`);
  }
};
```

Function Context

In non-strict mode, `this` in a standalone function refers to the global object (window in browsers):

```
function showThis() {
 console.log(this);
} // Shows window/global
```



Made with GAMMA

Arrow Functions: Syntax & Behaviour

1

Concise Syntax

Arrow functions provide a compact alternative to traditional function expressions:

```
// Traditional
const add = function(a, b) {
  return a + b;
};

// Arrow function
const add = (a, b) => a + b;
```

2

Lexical `this`

Arrow functions inherit `this` from their containing scope, solving a common problem in callbacks:

```
class Timer {
  constructor() {
    this.seconds = 0;
    // Arrow function preserves `this`
    setInterval(() => {
      this.seconds++;
    }, 1000);
  }
}
```

3

Limitations

Cannot be used with `new`, as constructors, or as methods where `this` binding is important:

```
// Invalid - no own `this`
const Person = (name) => {
  this.name = name;
};
const john = new Person('John'); // Error
```

Arrow functions are especially useful for short callbacks, array methods like `map` and `filter`, and preserving context in event handlers.

JavaScript Classes

Classes provide a clear syntax for object-oriented programming in JavaScript, though they're built on the same prototype-based inheritance system that has always existed in the language.

Class Declaration

```
class Person {
  constructor(name) {
    this.name = name;
  }

  greet() {
    return `Hello, I'm ${this.name}`;
  }

  static create(name) {
    return new Person(name);
  }
}
```

Inheritance

```
class Employee extends Person {
  constructor(name, role) {
    super(name); // Call parent constructor
    this.role = role;
  }

  greet() {
    return `${super.greet()}. I work as a ${this.role}`;
  }
}
```

Private class fields using the `#` prefix are now available in modern browsers, enabling true encapsulation in JavaScript class design.

Modules in JavaScript

Exporting

```
// Named exports
export const PI = 3.14159;
export function square(x) {
  return x * x;
}

// Default export
export default class Calculator {
  // Implementation
}
```

Importing

```
// Named imports
import { PI, square } from './math.js';

// Default import
import Calculator from './calculator.js';

// Mix of named and default
import Calculator, { PI } from './math.js';

// Import all as namespace
import * as MathUtils from './math.js';
```

Using Modules

```
// In HTML
<script type="module" src="app.js">
</script>

// In Node.js
// package.json: { "type": "module" }
import { readFile } from 'fs/promises';
```

ES Modules offer better performance through static analysis, allowing tree-shaking (dead code elimination) during bundling. They execute in strict mode by default and have their own scope, unlike scripts which share the global scope.

JSON: JavaScript Object Notation

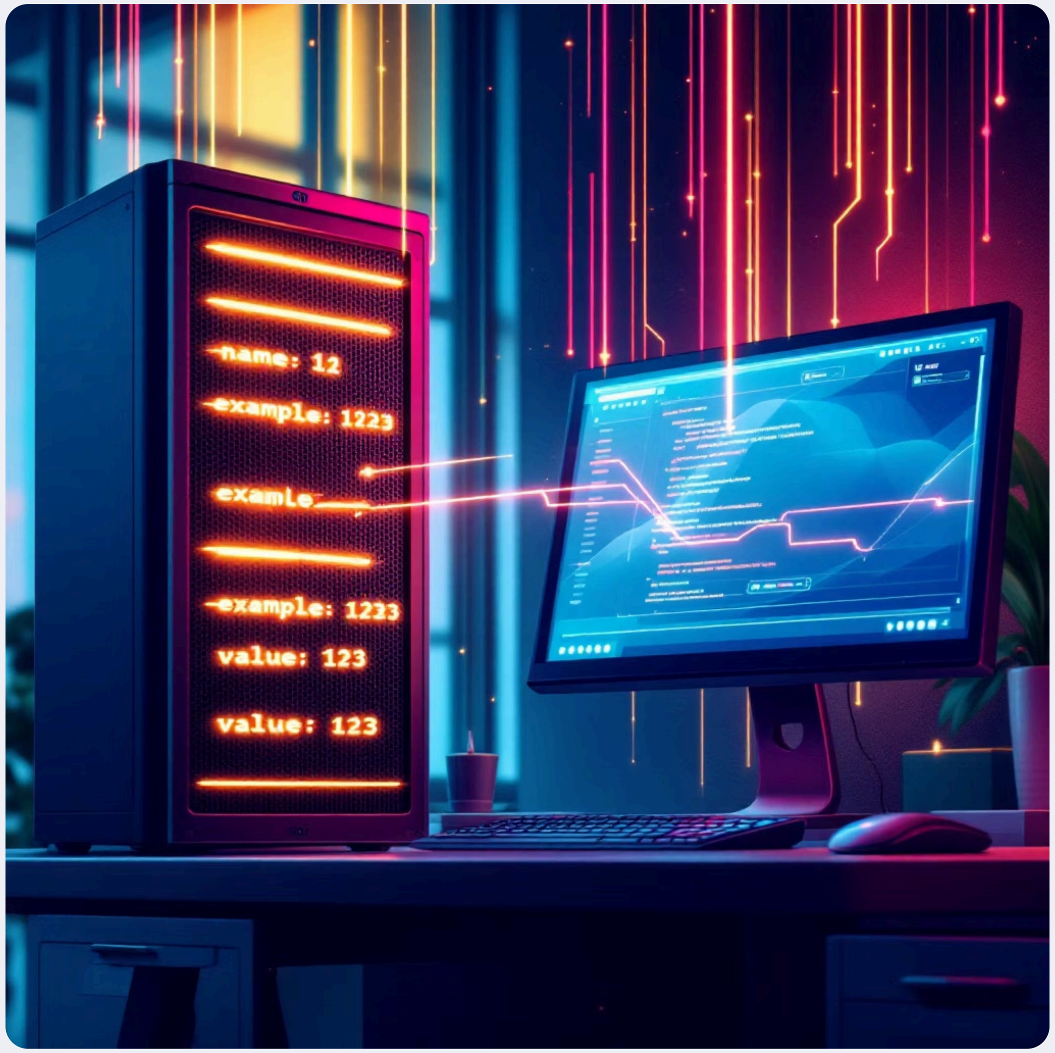
Working with JSON

Converting to JSON

```
const user = {
  name: 'Emma',
  age: 28,
  isAdmin: false
};

const json = JSON.stringify(user);
// '{"name":"Emma","age":28,"isAdmin":false}'
```

Parsing JSON

```
const jsonStr = '{"name":"Emma","age":28}';
const obj = JSON.parse(jsonStr);
console.log(obj.name); // 'Emma'
```



JSON supports object literals, arrays, strings, numbers, booleans, and null. It does not support functions, undefined, dates (stored as strings), or circular references.

For dates and special objects, use the optional replacer/reviver parameters of stringify/parse to customise serialisation behaviour.

Despite its name, JSON is language-independent and used across countless APIs, configuration files, and data exchange scenarios throughout the web ecosystem.

Callbacks: Handling Asynchronous Code

1 **The Callback Pattern**

A callback is a function passed as an argument to another function, to be executed after a task completes:

```
function fetchData(url, callback) {
  // Simulate network request
  setTimeout(() => {
    const data = { id: 1, name: 'Product' };
    callback(null, data);
  }, 1000);
}
```

2 **Using Callbacks**

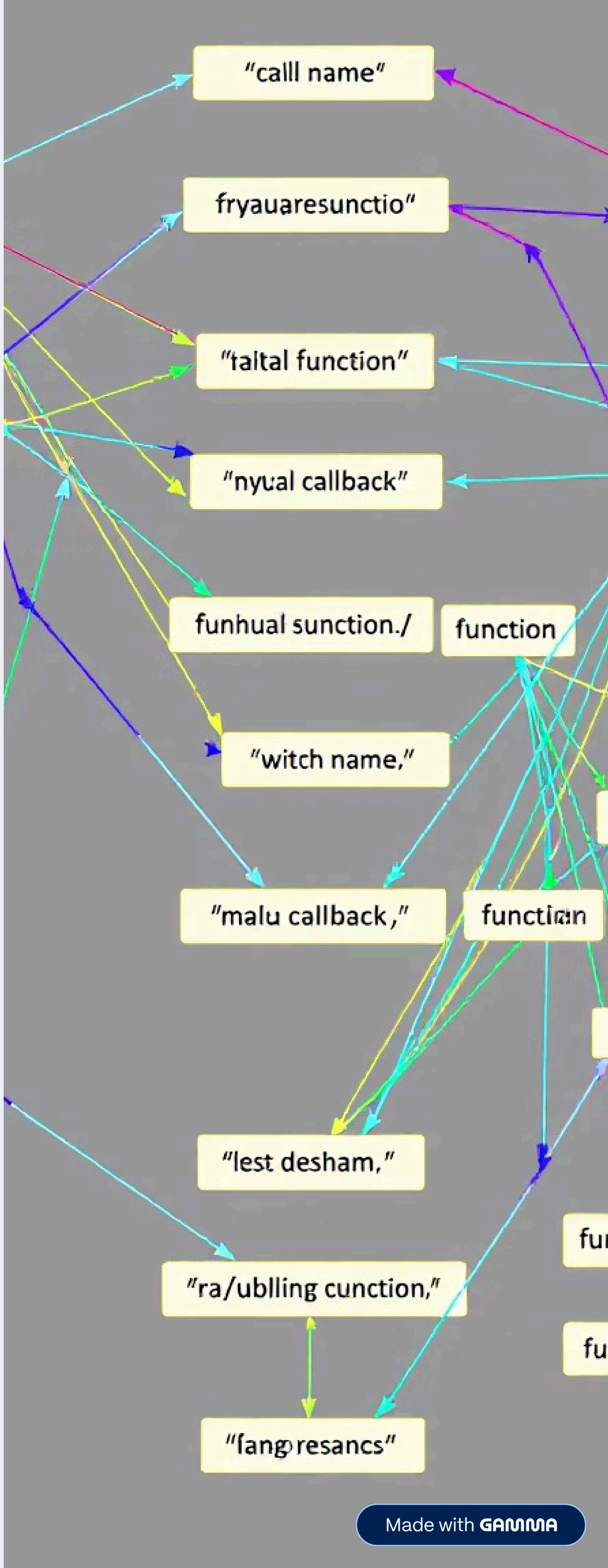
Callbacks enable non-blocking operations in JavaScript's single-threaded environment:

```
fetchData('api/products', (error, data) => {
  if (error) {
    console.error('Error:', error);
    return;
  }
  console.log('Data received:', data);
});
```

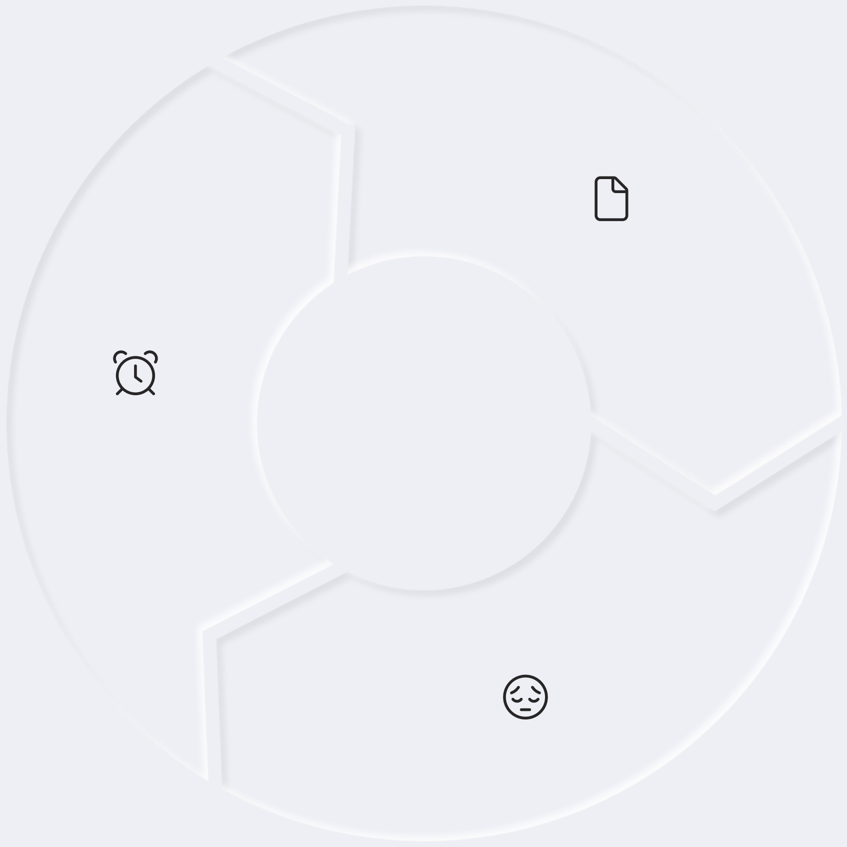
3 **Callback Hell**

Deeply nested callbacks create hard-to-maintain code:

```
fetchUser(userId, (error, user) => {
 if (error) return handleError(error);
 fetchOrders(user.id, (error, orders) => {
 if (error) return handleError(error);
 fetchProducts(orders[0].id, (error, products) => {
 // Deeper and deeper...
 });
 });
});
```



Promises: Cleaner Async Handling



Creating and Using Promises

```
// Creating a promise
function fetchData(url) {
  return new Promise((resolve, reject) => {
    // Async operation
    fetch(url)
      .then(response => {
        if (!response.ok) {
          throw new Error('Network error');
        }
        return response.json();
      })
      .then(data => resolve(data))
      .catch(error => reject(error));
  });
}

// Using a promise
fetchData('api/users')
  .then(users => fetchData(`api/users/${users[0].id}`))
  .then(user => console.log(user))
  .catch(error => console.error(error))
  .finally(() => console.log('Operation complete'));
```

Pending

Initial state before resolution or rejection. The promise is still processing.

Fulfilled

Operation completed successfully. The promise resolves with a value.

Rejected

Operation failed. The promise rejects with a reason (error).

Promise.all(), Promise.race(), Promise.allSettled(), and Promise.any() allow powerful combinations of multiple asynchronous operations.

Async/Await: Syntactic Sugar for Promises

1

Basic Syntax

```
// Async function declaration
async function fetchUserData() {
  // Await pauses execution until promise
resolves
  const response = await
fetch('/api/users');
  const data = await response.json();
  return data;
}

// Using the async function
fetchUserData()
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

2

Error Handling

```
async function fetchUserData() {
  try {
    const response = await
fetch('/api/users');
    if (!response.ok) {
      throw new Error('Network response
failed');
    }
    const data = await response.json();
    return data;
  } catch (error) {
    console.error('Error fetching user
data:', error);
    throw error; // Re-throw or handle
appropriately
  }
}
```

3

Parallel Execution

```
async function fetchAllData() {
 try {
 // Start all fetches in parallel
 const usersPromise = fetch('/api/users');
 const productsPromise =
fetch('/api/products');

 // Wait for both to complete
 const [usersResponse,
productsResponse] =
 await Promise.all([usersPromise,
productsPromise]);

 // Process results
 const users = await
usersResponse.json();
 const products = await
productsResponse.json();

 return { users, products };
 } catch (error) {
 console.error('Failed to fetch data:',
error);
 }
}
```

Introduction to React Commands

Project Setup

```
npx create-react-app my-app
cd my-app
npm start
```

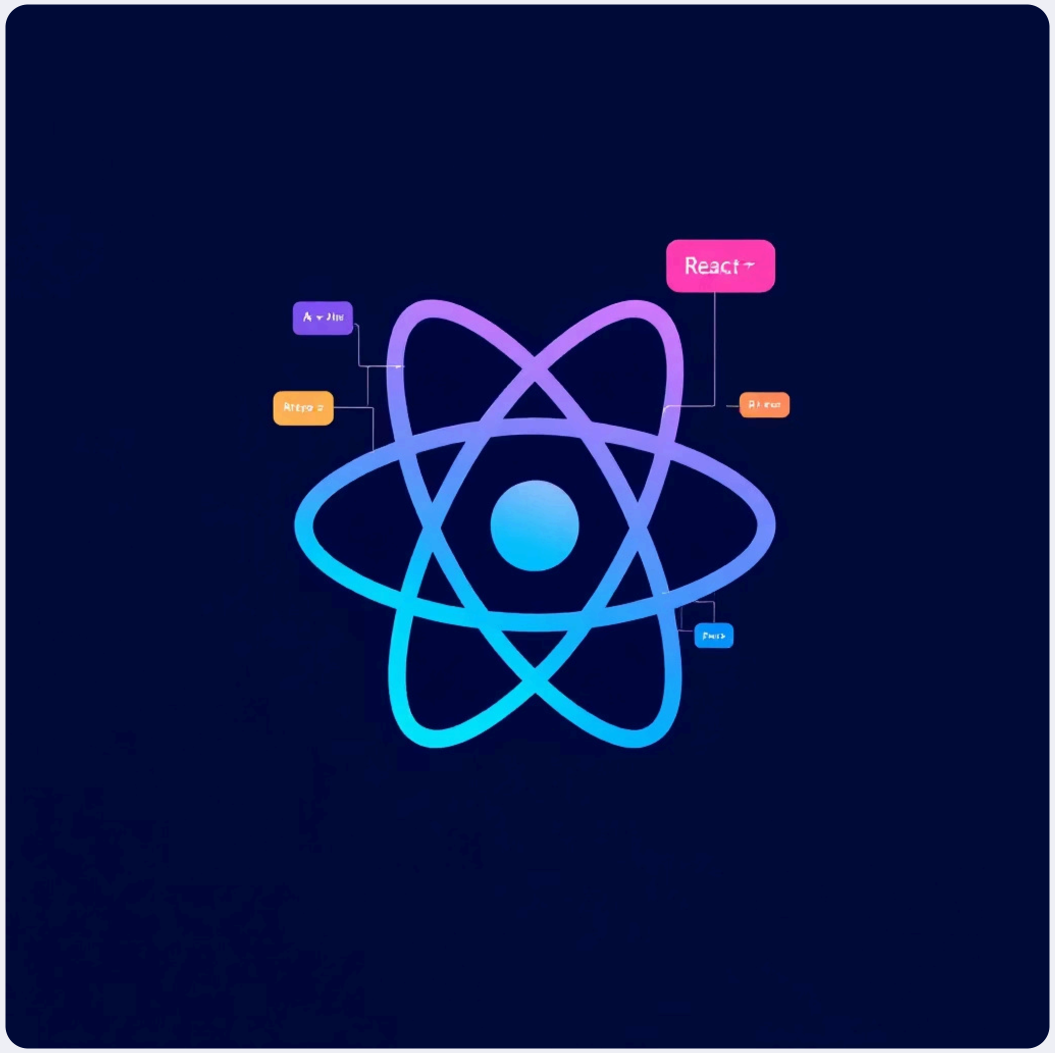
Component Creation

```
// Functional Component with Hooks
import React, { useState, useEffect } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `Count: ${count}`;
  }, [count]);

  return (
```

Count: {count}

setCount(count + 1)}> Increment

); }



Essential React Commands

- npm run build - Create production build
- npm test - Run test suite
- npm run eject - Eject from Create React App

Modern React Development

React's functional approach with hooks has largely replaced class components. The useState, useEffect, useContext, and useReducer hooks form the core of modern React state management and side effect handling.

Consider Next.js for production applications requiring server-side rendering and optimised routing.