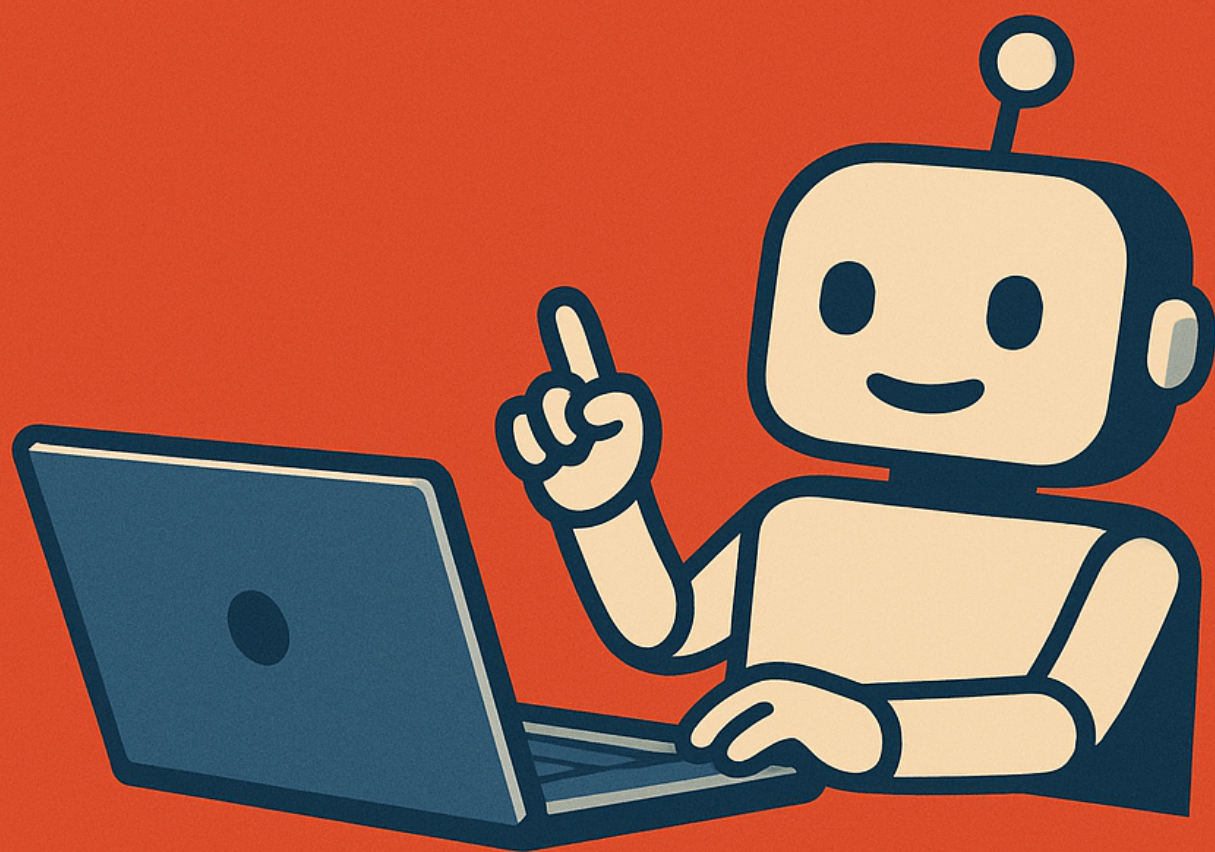


PROMPT TO BUILD

CREATE REAL APPS WITH AI



BRENT MILNE

Copyright © 2025 Brent Milne.

This book and the information contained herein are copyrighted by Brent Milne. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Disclaimer:

The information contained in this book is for informational purposes only. While the author has made every effort to ensure the accuracy of the information, the content should not be considered professional advice. The author and publisher assume no responsibility for errors or omissions or damages resulting from the use of the information contained herein. This book explores the use of AI tools for code generation. The author has made every effort to provide accurate and up-to-date information. However, AI technology is constantly evolving, and the results generated by AI tools can vary. The author and publisher are not responsible for the accuracy, reliability, or suitability of AI-generated code, and readers are advised to exercise caution and critical judgment when using such tools.

Chapter 1: The New Frontier - AI and Code Generation

Welcome to the New Frontier of Coding

Just as Leo Brodie's *Starting Forth* introduced a generation of programmers to the elegant simplicity and power of the Forth language, this book aims to guide you through a new frontier: the world of AI-assisted code generation. In an era where artificial intelligence is rapidly transforming every industry, software development is no exception. AI tools are no longer a futuristic concept; they are here, and they are changing how we write, debug, and even think about code.

This book is designed for anyone who wants to harness the immense power of AI to become a more efficient, productive, and innovative programmer. Whether you're a complete beginner taking your first steps into coding or an experienced developer looking to integrate cutting-edge tools into your workflow, you'll find a structured, hands-on approach to mastering the art of prompting AI for code.

What is AI-Assisted Code Generation?

At its core, AI-assisted code generation involves utilising artificial intelligence models, particularly large language models (LLMs), to assist developers in writing software. Think of these AI tools as intelligent assistants that can understand your natural language instructions (prompts) and translate them into functional code. They can:

- **Generate new code:** From simple functions to complex algorithms, based on your descriptions.
- **Complete code:** Suggesting the following lines of code as you type, much like an advanced autocomplete.
- **Refactor and optimise code:** Improving existing code for readability, performance, or adherence to best practices.
- **Debug and fix errors:** Identifying issues in your code and suggesting corrections.
- **Translate code:** Converting code from one programming language to another.
- **Generate documentation:** Creating comments, docstrings, and even complete documentation for your codebase.

AI models power these capabilities, which are trained on vast datasets of code and text, allowing them to learn patterns, syntax, and standard programming paradigms across multiple languages. When you provide a prompt, the AI leverages this learned knowledge to predict and generate relevant code.

The Benefits of AI-Assisted Coding

The integration of AI into the coding workflow offers several compelling advantages:

1. **Increased Productivity:** AI can automate repetitive tasks, generate boilerplate code, and quickly provide solutions to common problems, freeing up developers to focus on more complex and creative challenges.
2. **Faster Prototyping:** Rapidly generate initial code for new features or projects, accelerating the prototyping phase and enabling quicker iteration.
3. **Reduced Errors:** AI tools can detect syntax errors, logical flaws, and even security vulnerabilities early in the development process, resulting in more robust and reliable software.
4. **Learning and Exploration:** For beginners, AI can serve as a tutor, explaining code, suggesting alternative approaches, and providing examples. For experienced developers, it can help explore new libraries, frameworks, or programming languages more quickly.
5. **Consistency and Best Practices:** AI can be prompted to adhere to specific coding standards, style guides, and architectural patterns, promoting consistency across a codebase.
6. **Bridging Knowledge Gaps:** When working with unfamiliar APIs or complex domains, AI can provide relevant code snippets and explanations, reducing the time spent on research.

Limitations and Considerations

While incredibly powerful, AI-assisted code generation is not a magic bullet. It's crucial to understand its limitations:

1. **AI is a Tool, Not a Replacement:** AI assists, it does not replace the developer. Human oversight, critical thinking, and understanding of the generated code are paramount. You are still the architect and the quality assurance.
2. **Context is Key:** The quality of AI-generated code heavily depends on the clarity and completeness of your prompts. Ambiguous or insufficient prompts will lead to suboptimal or incorrect results.
3. **Potential for Errors:** AI can generate incorrect, inefficient, or insecure code. Continually review, test, and validate AI-generated code before integrating it into your

projects.

4. **Security and Privacy:** Be mindful of the data you share with AI tools, especially if working with proprietary or sensitive code. Some tools offer local execution options for enhanced privacy and security.
5. **Bias and Hallucinations:** AI models can inherit biases from their training data, and they can sometimes "hallucinate" or confidently present incorrect information as fact.
6. **Evolving Landscape:** The field of AI is evolving at a breakneck pace. Tools and best practices will continue to grow, requiring continuous learning and adaptation.

Why This Book? The Importance of Effective Prompting

Just as *Starting Forth* taught people to think in terms of "words" and how to combine them to build complex programs, this book will teach you to think in terms of "prompts" – the precise instructions you give to an AI. The quality of the code you get from an AI is directly proportional to the quality of the prompt you provide.

This book will guide you through:

- **Understanding AI's capabilities:** What can AI realistically do for you in coding?
- **Crafting effective prompts:** Learning the syntax, structure, and strategies for clear, concise, and comprehensive instructions.
- **Iterating and refining:** How to work with AI in a conversational manner to improve generated code.
- **Integrating AI into your workflow:** Practical ways to use AI tools in your daily development tasks.
- **Best practices and ethical considerations:** Ensuring you use AI responsibly and effectively.

We will explore these concepts using **JavaScript** as our primary language for examples. JavaScript's ubiquity, accessibility, and versatility make it an ideal choice for demonstrating AI-assisted code generation across various applications, from web development to standalone tools.

A Nod to "Starting Forth"

Starting Forth was revolutionary in its approach to teaching programming. It broke down complex concepts into digestible pieces, used explicit language, and reinforced learning with practical examples and even a touch of humour through its iconic illustrations. We aim to capture that spirit here.

Each chapter will introduce new prompting techniques and coding concepts, building upon the knowledge gained in previous chapters. You'll find clear explanations, hands-on examples, and exercises designed to solidify your understanding. Just as Forth encouraged a disciplined yet creative approach to programming, so too will this book promote a thoughtful and iterative approach to AI prompting.

Welcome aboard. Let's start prompting for code!

////////////////////////////////////

References:

- Brodie, Leo. *Starting Forth*. Prentice-Hall, 1981. (Available online: <http://www.forth.com/starting-forth/>)
- Qodo.ai. "15 Best AI Coding Assistant Tools in 2025." <https://www.qodo.ai/blog/best-ai-coding-assistant-tools/>
- Pieces.app. "10 Best AI code generators in 2025 [Free & Paid]." <https://pieces.app/blog/9-best-ai-code-generation-tools>



Chapter 2: Setting Up Your AI Coding Workbench

Choosing Your Language: JavaScript

Before we dive into the exciting world of AI-assisted code generation, we need to establish our foundational programming language. Just as *Starting Forth* focused on Forth, this book will centre on **JavaScript**. This means plain JavaScript, without any additional libraries or frameworks, such as jQuery, React, or Angular. This choice is deliberate and offers several key advantages for beginners and for demonstrating AI prompting:

- **Ubiquity:** JavaScript is the language of the web. Every modern web browser has a built-in JavaScript engine, meaning you can write and run JavaScript code directly in your browser without needing to install complex development environments.
- **Accessibility:** It's free to use and learn. All you need is a text editor and a web browser, both of which are readily available on almost any computer.
- **Versatility:** While primarily known for web development, JavaScript, especially with Node.js (which we'll explore later), can be used for server-side programming, desktop applications, mobile apps, and even embedded systems. This broad applicability makes it an excellent choice for demonstrating the power of AI-generated code across different domains.
- **Readability for AI:** The vast amount of JavaScript code available online makes it a rich training ground for AI models. This means AI tools are generally very good at understanding JavaScript prompts and generating accurate, idiomatic JavaScript code.
- **Foundation for Future Learning:** Mastering JavaScript provides a strong foundation for learning more advanced JavaScript libraries and frameworks later on. It teaches you the core concepts without the added complexity of external tools.

Understanding AI Coding Tools: Your Intelligent Assistants

Throughout this book, we'll be interacting with AI coding tools. These are software applications or services that use artificial intelligence to assist developers in various coding tasks. They come in many forms, but generally fall into these categories:

1. **Code Autocompletion and Suggestion:** These tools integrate directly into your code editor (like Visual Studio Code, which we'll set up in the next chapter) and provide real-time suggestions as you type. Think of them as a super-powered autocomplete that understands context and can suggest entire lines or blocks of code.
 - *Examples:* GitHub Copilot, Codeium, Tabnine.
2. **Code Generation from Natural Language:** This is where you describe what you want in plain English (or another human language), and the AI generates the corresponding code. This is the primary focus of our book.
 - *Examples:* ChatGPT, Gemini, specialised code generation platforms.
3. **Code Refactoring and Optimisation:** These tools can analyse your existing code and suggest improvements for readability, performance, and adherence to best practices.
4. **Debugging and Error Fixing:** Some AI tools can help identify bugs in your code and even suggest fixes.
5. **Code Translation:** AI can translate code from one programming language to another.
6. **Documentation Generation:** AI can generate comments, docstrings, and complete documentation for your codebase.

How They Work (Simplified)

Most of these tools are powered by large language models (LLMs) that have been trained on massive datasets of text and code. When you provide a prompt, the AI uses its training to predict the most likely and relevant code or text to generate. It's like having an incredibly knowledgeable coding assistant at your fingertips, ready to help you write code, explain concepts, or even brainstorm solutions.

Your Role in the AI-Assisted Workflow

It's crucial to understand that AI coding tools are **assistants**, not replacements for human developers. Your role is to:

- **Define the Problem:** Clearly articulate what you want the code to do.
- **Craft Effective Prompts:** Learn how to communicate your needs to the AI in a way it understands best.
- **Review and Verify:** Always examine the AI-generated code for correctness, efficiency, and security. AI can make mistakes or generate suboptimal solutions.
- **Integrate and Refine:** Incorporate the AI-generated code into your project and refine it as needed.

- **Debug and Test:** Even with AI assistance, thorough testing and debugging are essential.

Think of it as a collaboration. You provide the high-level direction and quality control, and the AI handles the heavy lifting of code generation. This partnership allows you to be significantly more productive and tackle more ambitious projects.

Your First AI Interaction: The "Hello, World!" of Prompting

Let's start with the simplest possible interaction: asking the AI to generate a classic "Hello, World!" program in JavaScript. This will demonstrate the basic flow of prompting.

Prompt:

```
Write a simple JavaScript code snippet that prints "Hello, World!" to the console.
```

AI-Generated Code (Expected):

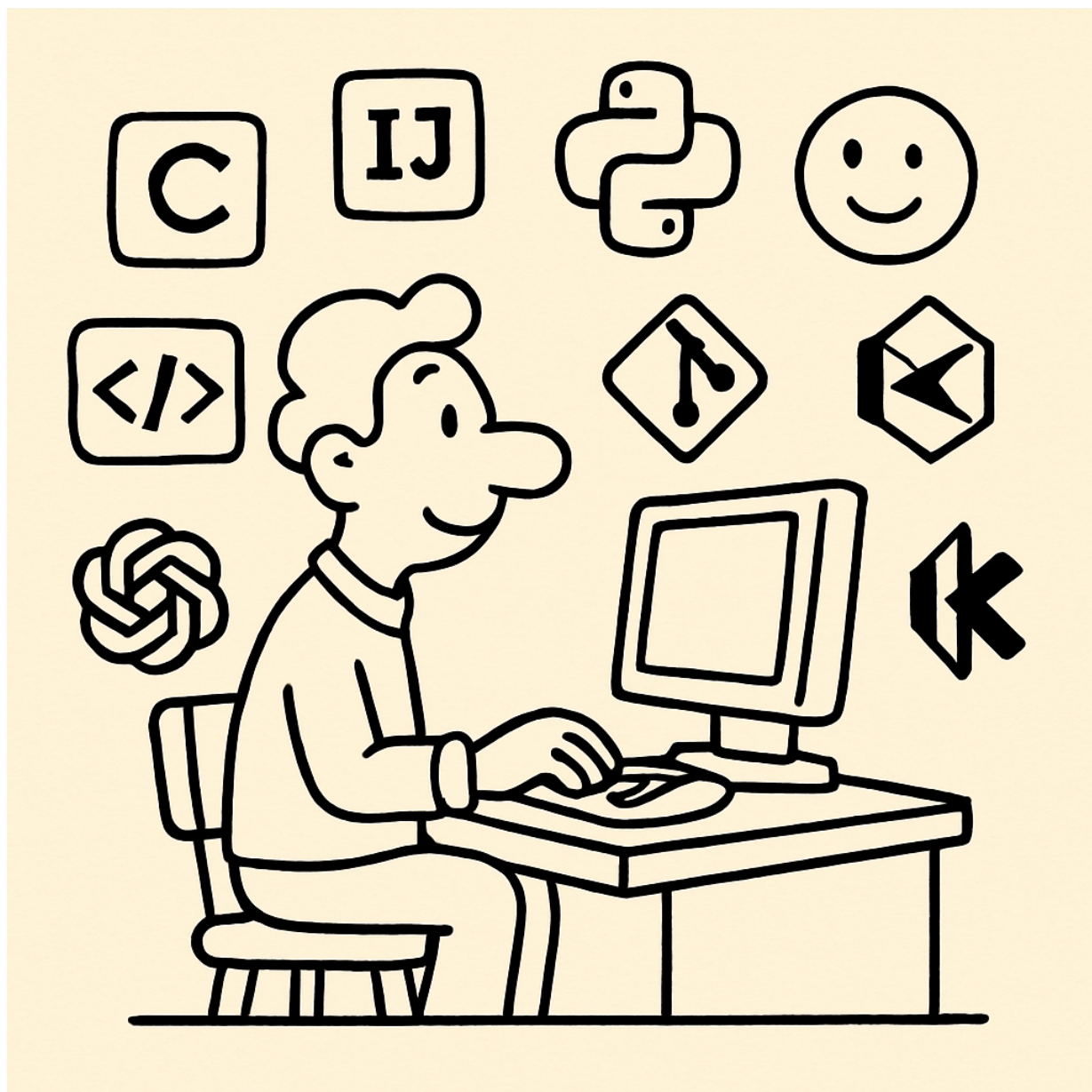
```
console.log("Hello, World!");
```

- **Analysis:** The AI correctly interprets your request and provides the standard JavaScript command for printing to the console. This is the fundamental interaction we'll build upon throughout the book.

In the next chapter, we'll set up your development environment, focusing on Visual Studio Code, which will be your primary tool for writing and running JavaScript code, as well as interacting with AI assistants.

References:

- [MDN Web Docs: JavaScript](#)
- [Node.js Official Website](#)
- [What is Vanilla JavaScript?](#)



Chapter 3: Setting Up Your AI Coding Workbench

Your Digital Workshop: Visual Studio Code and Beyond

Just as a craftsman needs a well-equipped workshop, a programmer needs a robust and efficient development environment. For JavaScript development, especially when working with AI-assisted code generation, **Visual Studio Code (VS Code)** is an excellent choice. It's free, open-source, highly customisable, and packed with features that will make your coding journey smoother. This chapter will guide you through setting up VS Code and its essential extensions, transforming your computer into a powerful AI coding workbench.

Why Visual Studio Code?

VS Code has become the go-to code editor for millions of developers for several reasons:

- **Lightweight and Fast:** It starts quickly and runs efficiently, even on less powerful machines.
- **Free and Open Source:** Accessible to everyone, with a large and active community.
- **Cross-Platform:** Available on Windows, macOS, and Linux.
- **Extensible:** A vast marketplace of extensions allows you to add features for almost any programming language or task.
- **Built-in Features:** Includes integrated terminal, Git control, and a powerful debugger.
- **IntelliSense:** Provides intelligent completions based on variable types, function definitions, and imported modules, making coding faster and reducing errors.

Step-by-Step Setup: Getting Started with VS Code

Step 1: Download and Install VS Code

1. **Go to the Official Website:** Open your web browser and navigate to code.visualstudio.com.
2. **Download:** Click the prominent "Download" button for your operating system (Windows,

macOS, or Linux).

3. **Install:** Once the download is complete, run the installer. Follow the on-screen prompts. For most users, the default options are fine. On macOS, you might just drag the application to your Applications folder.

Step 2: Install Essential Extensions

Extensions add powerful capabilities to VS Code. Here are a few crucial ones for JavaScript development and AI-assisted coding:

1. **Open the Extensions View:** In VS Code, click on the Extensions icon in the Activity Bar on the side (it looks like four squares, one of which is detached) or press `Ctrl+Shift+X` (Windows/Linux) or `Cmd+Shift+X` (macOS).
2. **Search and Install:** In the search bar at the top of the Extensions view, type the name of the extension and click "Install" next to the correct one.
 - **Prettier - Code formatter:** This extension automatically formats your code to ensure consistency and readability. It saves you time and helps maintain a clean codebase.
 - **Why it helps:** AI-generated code might not always follow your preferred formatting. Prettier ensures everything looks neat.
 - **ESLint:** A linter that analyses your code for potential errors, bugs, stylistic issues, and suspicious constructs. It helps you write higher-quality JavaScript.
 - **Why it helps:** Catches common JavaScript mistakes and enforces best practices, complementing AI-generated code by ensuring it meets quality standards.
 - **Live Server:** Launches a local development server with a live reload feature for static and dynamic pages. Any changes you make to your HTML, CSS, or JavaScript files will automatically update in your browser.
 - **Why it helps:** Speeds up your development workflow by showing changes instantly without manual refreshing.
 - **Node Essentials:** Installs Node.js in your development environment
 - **Why it helps** You can write and test Node.js files locally.
 - **GitHub Copilot (or similar AI coding assistant):** While this book teaches you *how* to prompt, having an AI coding assistant integrated directly into your editor can provide real-time suggestions and completions. (Note: Some AI coding assistants may require a subscription or account setup.)

- **Why it helps:** Provides immediate feedback and code suggestions as you type, making the prompting process more interactive.

Step 3: Configure VS Code (Optional but Recommended)

VS Code is highly configurable. You can access settings by going to

`File > Preferences > Settings` (Windows/Linux) or `Code > Settings > Settings` (macOS) or by pressing `Ctrl+,` (Windows/Linux) or `Cmd+,` (macOS).

- **Default Formatter:** Set Prettier as your default formatter.
 1. Search for "Default Formatter".
 2. Select "Prettier - Code formatter" from the dropdown.
 3. Enable "Format On Save" by searching for it and checking the box.
- **Auto Save:** Ensure your work is saved automatically.
 1. Search for "Auto Save".
 2. Select "onFocusChange" or "afterDelay" from the dropdown.

Your First JavaScript File in VS Code

Let's create a simple JavaScript file to test your setup.

1. **Create a New Folder:** On your desktop or in your documents, create a new folder named `my-first-js-project`.
2. **Open Folder in VS Code:** In VS Code, go to `File > Open Folder...` and select the `my-first-js-project` folder you just created.
3. **Create a New File:** In the VS Code Explorer (the left sidebar), click the "New File" icon (a blank page with a plus sign) and name the file `app.js`.
4. **Write Some Code:** Type the following JavaScript code into `app.js`:

```
// app.js
console.log("Hello, AI-Assisted Coding!");

function greet(name) {
  return `Hello, ${name}!`;
}

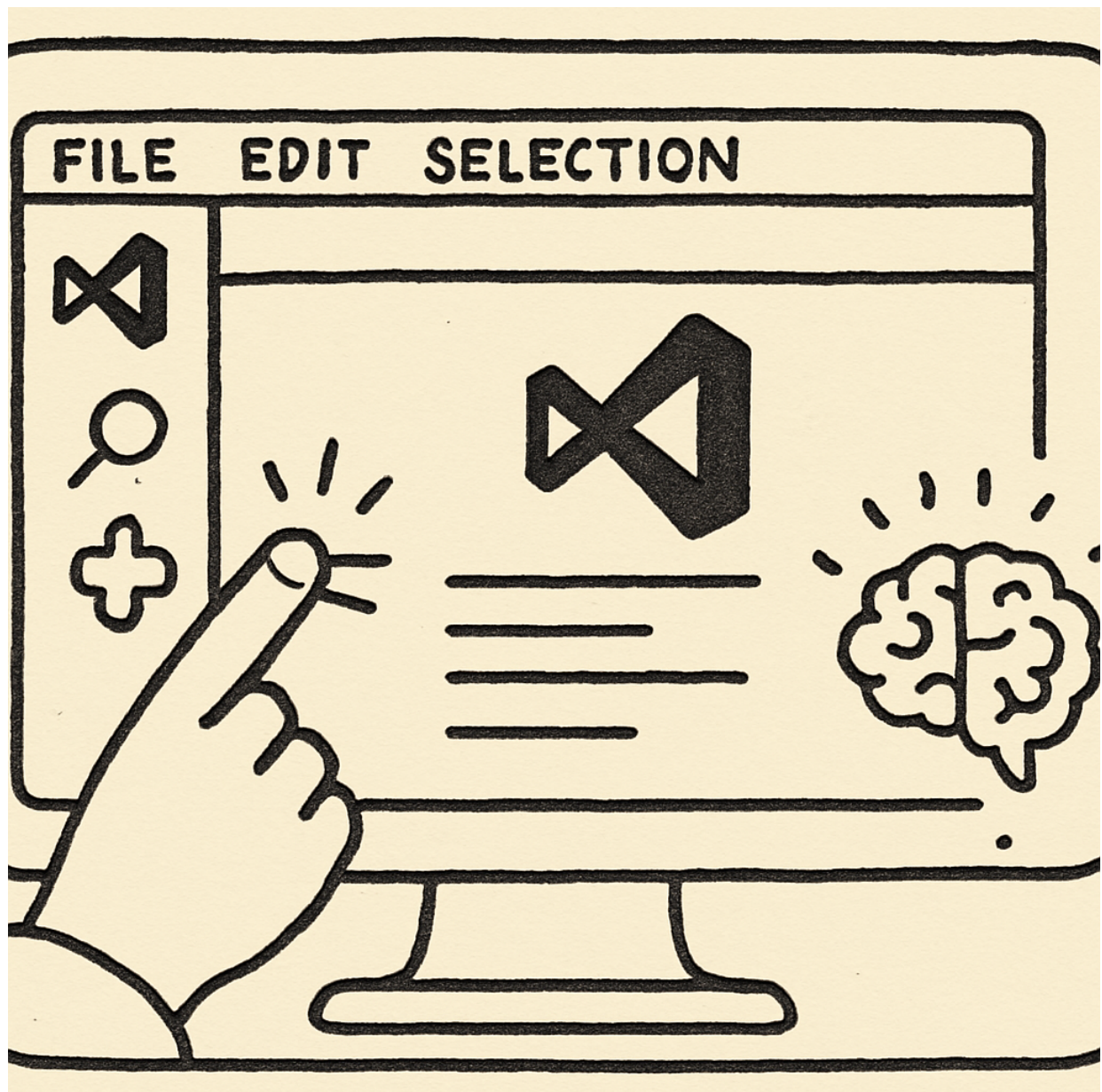
const message = greet("Developer");
console.log(message);
```

5. **Run in Terminal:**

- ```
Hello, AI-Assisted Coding!
Hello, Developer!
```

With Visual Studio Code set up and essential extensions installed, you now have a robust and comfortable environment for writing, testing, and debugging your JavaScript code. This workbench will be your primary tool as you continue to explore the exciting world of AI-assisted development. In the next chapter, we will dive deeper into the language of AI itself: prompt engineering.

- [Visual Studio Code Official Website](#)
- [VS Code Docs: User Guide](#)
- [Prettier Extension](#)
- [ESLint Extension](#)
- [Live Server Extension](#)





# Chapter 4: The Language of AI - Crafting Effective Prompts

## Speaking to the Machine: The Art of Prompt Engineering

---

Just as Forth relies on a precise vocabulary of "words" to build programs, communicating effectively with AI for code generation requires a precise "language" of its own: **prompt engineering**. This is the art and science of crafting inputs (prompts) that elicit the desired output from an AI model. A well-engineered prompt can transform a vague idea into functional code, while a poorly designed one can lead to irrelevant, incorrect, or incomplete results.

This chapter introduces the fundamental principles of prompt engineering for code generation. We will learn how to structure your prompts, provide necessary context, and guide the AI towards generating accurate and valid JavaScript code.

## Core Principles of Effective Prompts

---

Think of your AI assistant as an incredibly knowledgeable but literal intern. It will do exactly what you tell it, but it won't necessarily infer what you *meant* if your instructions are unclear. Here are the core principles:

1. **Be Clear and Specific:** Avoid ambiguity. State exactly what you want the AI to do, what inputs it will receive, and what output you expect.
2. **Provide Context:** Provide the AI with sufficient background information. What is the purpose of the code? What problem is it solving? What existing code does it need to integrate with?
3. **Define Constraints and Requirements:** Specify any limitations, such as the programming language, specific libraries to use or avoid, performance considerations, or coding style guidelines.
4. **Break Down Complex Tasks:** For larger problems, break them into smaller, manageable steps. You can prompt the AI for each step individually or guide it through a sequence of operations.
5. **Iterate and Refine:** Prompting is rarely a one-shot process. Expect to refine your

prompts based on the AI's initial responses. Think of it as a conversation.

## Anatomy of a Good Code Generation Prompt

---

A good prompt for code generation typically includes several key components:

- **Role/Persona (Optional but Recommended):** Tell the AI what role it should adopt (e.g., "Act as a senior JavaScript developer," "You are a security expert"). This can influence the style and focus of its response.
- **Task/Goal:** Clearly state the objective. What kind of code do you want? (e.g., "Write a function," "Generate an HTML structure," "Create a CSS style").
- **Input Description:** Describe any data or parameters the code will receive. Provide examples if possible.
- **Output Description:** Specify the expected format and content of the output. Do you need a function, a class, a whole file, or just a snippet? What should the return value be?
- **Constraints/Conditions:** Any rules, limitations, or specific requirements. (e.g., "Use JavaScript," "Do not use jQuery," "Ensure it's responsive," "Handle errors gracefully").
- **Examples (Few-Shot Prompting):** Providing examples of desired input-output pairs can significantly improve the quality of the AI's response, especially for complex transformations or specific formatting needs.

## Basic Prompting Examples

---

Let's apply these principles with some simple JavaScript examples.

### Example 1: Generating a Simple Function

**Goal:** A function that adds two numbers.

**Prompt:**

```
Write a JavaScript function called `addNumbers` that takes two arguments
```

**AI-Generated Code (Expected):**

```
function addNumbers(a, b) {
 return a + b;
}
```

- **Analysis:** This is a straightforward request. The AI correctly interprets your request and provides the standard JavaScript command for printing to the console.

## Example 2: Adding Input Validation

**Goal:** The `addNumbers` function should only add if the inputs are numbers.

**Prompt:**

```
Modify the `addNumbers` JavaScript function. It should now check if both `a` and `b` are numbers. If they are not, it should return an error message, such as "Invalid input: Both arguments must be numbers."
```

**AI-Generated Code (Expected):**

```
function addNumbers(a, b) {
 if (typeof a !== 'number' || typeof b !== 'number') {
 return "Invalid input: Both arguments must be numbers.";
 }
 return a + b;
}
```

- **Analysis:** By adding an explicit constraint ( `check if both a and b are numbers` ) and specifying the error message, the AI incorporates the validation logic. This demonstrates how iterative prompting can refine code.

## Example 3: Generating an Array Operation

**Goal:** A function to find the maximum number in an array.

**Prompt:**

```
Write a JavaScript function `findMax` that takes an array of numbers as input and returns the largest number in the array. Assume the input array will always contain numbers.
```

**\*\*AI-Generated Code (Expected):**

```
function findMax(numbers) {
 if (numbers.length === 0) {
 return undefined; // Or throw an error, depending on desired behavior
 }
 let max = numbers[0];
 for (let i = 1; i < numbers.length; i++) {
 if (numbers[i] > max) {
 max = numbers[i];
 }
 }
 return max;
}
```

- **Analysis:** The AI provides a correct implementation. Notice it also implicitly handles the edge case of an empty array, which is good practice.

## Providing Context: The Key to Relevant Code

---

AI models perform better when they understand the context in which the code will be used. This helps them generate more relevant and integrated solutions.

### Example 4: Context for HTML Manipulation

**Goal:** Add a new list item to an existing HTML unordered list.

**Prompt:**

```
I have an HTML page with an unordered list that has the ID "myList".
Write a JavaScript function `addListItem` that takes a string `itemText`
and adds a new list item () with that text to the "myList" list.
```

**AI-Generated Code (Expected):**

```
function addListItem(itemText) {
 const ul = document.getElementById("myList");
 if (ul) {
 const li = document.createElement("li");
 li.textContent = itemText;
 ul.appendChild(li);
 } else {
 console.error("Element with ID 'myList' not found.");
 }
}
```

- **Analysis:** By knowing the ID of the HTML element, the AI correctly uses

`document.getElementById` and `appendChild` to manipulate the DOM (Document Object Model). It also includes basic error handling if the element isn't found.

## Defining Constraints and Requirements

---

Explicitly stating constraints helps the AI narrow down its options and produce code that meets your specific needs.

### Example 5: Performance Constraint

**Goal:** A function to sum large numbers efficiently.

**Prompt:**

```
Write a JavaScript function `sumLargeNumbers` that takes an array of numbers. This array can be very large. Optimise the function for optimal performance, avoiding unnecessary iterations and complex operations. Return the total sum.
```

**AI-Generated Code (Expected):**

```
function sumLargeNumbers(numbers) {
 let sum = 0;
 for (let i = 0; i < numbers.length; i++) {
 sum += numbers[i];
 }
 return sum;
}

// Or, for a more modern approach that might be slightly less performant
// for *extremely* large arrays
// but is often more readable:
// function sumLargeNumbers(numbers) {
// return numbers.reduce((accumulator, currentValue) => accumulator +
// currentValue, 0);
// }
```

- **Analysis:** The AI provides a simple `for` loop, which is generally the most performant way to sum numbers in a large array in JavaScript, demonstrating its understanding of the "optimise for performance" constraint. It might also offer `reduce` as an alternative, highlighting trade-offs.

## Conclusion: Your First Steps in Prompt



# Engineering

---

You've now taken your first steps into the crucial skill of prompt engineering. By being clear, providing context, and defining constraints, you can significantly improve the quality and relevance of the code generated by AI. Remember, the AI is a powerful tool, but it's your ability to communicate effectively with it that will truly unlock its potential. In the next chapter, we'll move from simple functions to building more complex structures.

---

## References:

- [Prompt Engineering Guide: Introduction](#)
- [OpenAI: Best practices for prompt engineering with OpenAI API](#)
- [MDN Web Docs: Working with objects](#)



# Chapter 5: From Zero to Function - Building Blocks of Code

## The Smallest Units: Functions and Variables

---

In *Starting Forth*, people learned about "words" – the fundamental building blocks of the Forth language. In JavaScript, our primary building blocks are **variables** and **functions**. Variables are like containers that hold information, and functions are reusable blocks of code that perform specific tasks. Understanding how to prompt AI to generate these effectively is your next crucial step.

This chapter will guide you through prompting AI to create variables, define functions, and understand their fundamental interactions. We will start simple and gradually build up to more complex function structures.

## Variables: Storing Information

---

Variables are essential for storing data that your program needs to work with. In JavaScript, you declare variables using `let`, `const`, or (less commonly now) `var`.

- `let` : Used for variables whose values might change.
- `const` : Used for variables whose values will not change (constants).
- `var` : An older keyword, generally avoided in modern JavaScript due to its scope rules.

### Example 1: Declaring Variables

**Goal:** Declare two variables to store a user's name and age.

**Prompt:**

```
Declare two JavaScript variables: one named `userName` to store a string value "Alice", and another named `userAge` to store a number value 30. Use modern JavaScript syntax.
```

**AI-Generated Code (Expected):**

```
const userName = "Alice";
let userAge = 30;
```

- **Analysis:** The AI correctly uses `const` for a value that is unlikely to change (name) and `let` for one that might (age), demonstrating an understanding of modern JavaScript conventions.

## Example 2: Performing Operations with Variables

**Goal:** Calculate the area of a rectangle using variables for length and width.

**Prompt:**

```
Write JavaScript code to declare two variables, `length` set to 10 and
`width` set to 5. Then, calculate their product and store it in a new
variable called `area`. Finally, print the `area` to the console.
```

**AI-Generated Code (Expected):**

```
const length = 10;
const width = 5;
const area = length * width;
console.log("The area is: " + area);
// Expected output: The area is: 50
```

- **Analysis:** The AI correctly performs arithmetic operations and demonstrates how to output results, a fundamental programming concept.

## Functions: Reusable Blocks of Code

Functions are the workhorses of JavaScript. They enable you to encapsulate a specific task into a reusable block, making your code more organised, efficient, and easier to debug. Functions can take inputs (arguments) and produce outputs (return values).

### Example 3: A Simple Function

**Goal:** Create a function that greets a user by name.

**Prompt:**

```
Write a JavaScript function named `greetUser` that takes one argument,
`name`, and returns a string like "Hello, [name]!".
```

### AI-Generated Code (Expected):

```
function greetUser(name) {
 return `Hello, ${name}!`; // Using template literals for easy string fo
}

// Example usage:
// console.log(greetUser("Bob")); // Expected output: Hello, Bob!
```

- **Analysis:** The AI correctly defines a function, uses a parameter, and returns a formatted string. The use of template literals (``) is a modern JavaScript feature that makes string concatenation cleaner.

## Example 4: Function with Multiple Arguments

**Goal:** A function to calculate the average of three numbers.

**Prompt:**

```
Create a JavaScript function called `calculateAverage` that accepts
three numbers as arguments (`num1`, `num2`, `num3`) and returns their
average.
```

### AI-Generated Code (Expected):

```
function calculateAverage(num1, num2, num3) {
 return (num1 + num2 + num3) / 3;
}

// Example usage:
// console.log(calculateAverage(10, 20, 30)); // Expected output: 20
```

- **Analysis:** This demonstrates how functions can handle multiple inputs and perform calculations based on them.

## Example 5: Functions Calling Other Functions

Just as Forth words can be combined to create new words, JavaScript functions can call other functions. This is a powerful way to build complex logic from simpler, reusable parts.

**Goal:** Calculate the total cost, including tax, using separate functions for subtotal and tax calculation.

**Prompt:**

First, write a JavaScript function `calculateSubtotal` that takes an array of prices and returns their sum.

Second, write a JavaScript function `calculateTax` that takes a subtotal and a tax rate (e.g., 0.05 for 5%) and returns the tax amount.

Finally, write a function `calculateTotalCost` that takes an array of prices and a tax rate. This function should use `calculateSubtotal` and `calculateTax` to compute and return the final total cost.

### AI-Generated Code (Expected):

```
function calculateSubtotal(prices) {
 let subtotal = 0;
 for (let i = 0; i < prices.length; i++) {
 subtotal += prices[i];
 }
 return subtotal;
}

function calculateTax(subtotal, taxRate) {
 return subtotal * taxRate;
}

function calculateTotalCost(prices, taxRate) {
 const subtotal = calculateSubtotal(prices);
 const taxAmount = calculateTax(subtotal, taxRate);
 return subtotal + taxAmount;
}

// Example usage:
// const itemPrices = [100, 50, 25];
// const salesTaxRate = 0.08; // 8%
// console.log(calculateTotalCost(itemPrices, salesTaxRate));
// Expected output: (100+50+25) + (100+50+25)*0.08 = 175 + 14 = 189
```

- **Analysis:** This example beautifully illustrates modularity. The AI correctly breaks down the problem into smaller, interconnected functions, demonstrating a fundamental principle of good software design.

## Conclusion: Your Growing Vocabulary

---

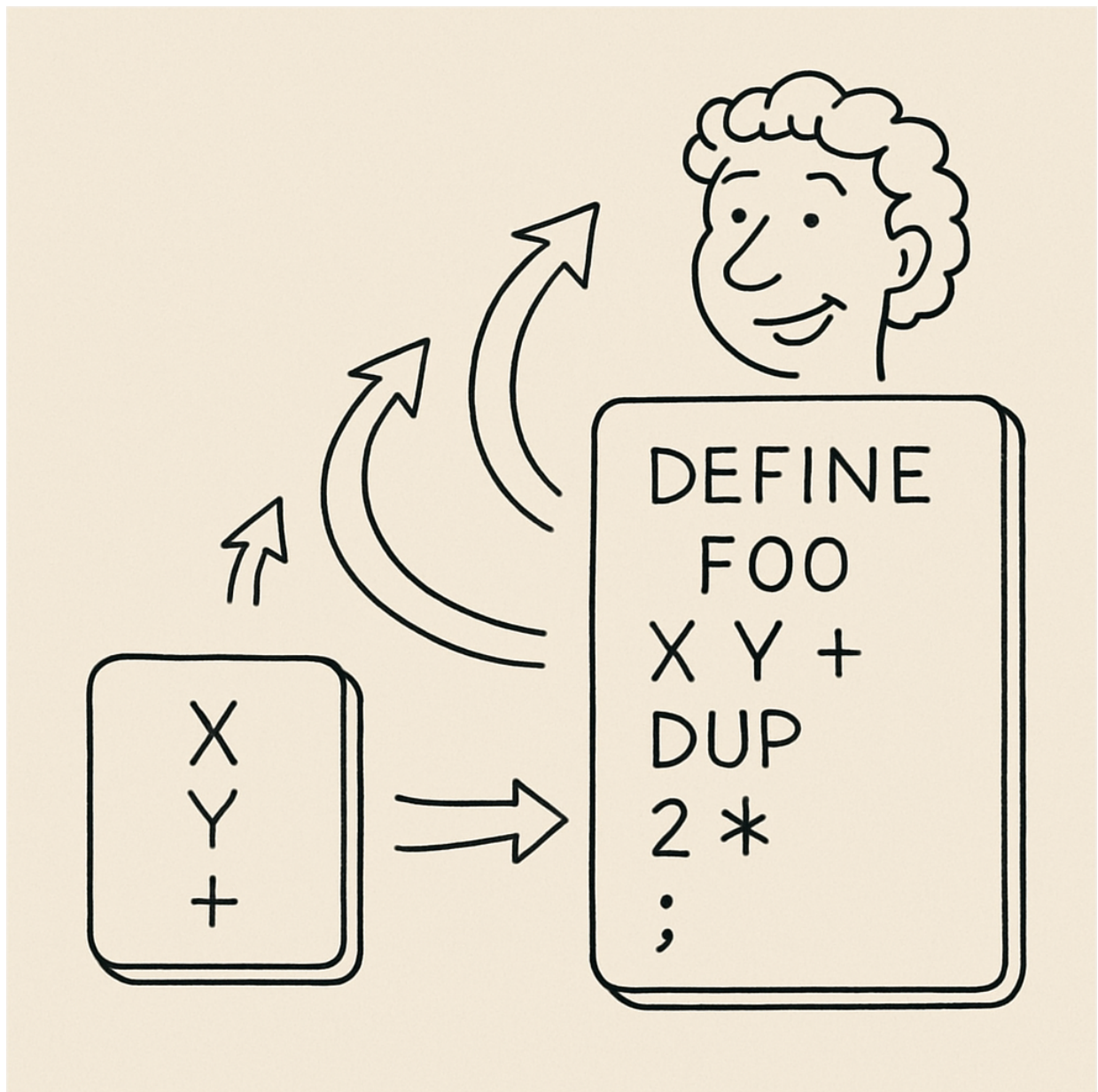
You've now mastered the basic "words" of JavaScript: variables and functions. You can prompt AI to create them, manipulate them, and combine them to perform more complex tasks. This foundational knowledge is crucial for building any JavaScript application. In the

next chapter, we will explore how to work with collections of data, such as lists and dictionaries.

---

### References:

- [MDN Web Docs: Variables](#)
- [MDN Web Docs: Functions](#)
- [JavaScript.info: Functions](#)





# Chapter 6: Working with Data - Prompting for Structures and Manipulation

## Organising Information: Arrays and Objects

---

Just as Forth allowed you to manipulate data on the stack, JavaScript provides powerful ways to organise and manipulate collections of information. The two most fundamental data structures in JavaScript are **arrays** (ordered lists of items) and **objects** (collections of key-value pairs, similar to dictionaries). Understanding how to prompt AI to work with these structures is crucial for building any meaningful application.

This chapter will guide you through prompting AI to create, access, and manipulate data within arrays and objects. We will cover everyday operations and demonstrate how AI can assist in handling complex data scenarios.

## Arrays: Ordered Collections

---

Arrays are used to store multiple values in a single variable. Each item in an array has an index, starting from 0.

### Example 1: Creating and Accessing Arrays

**Goal:** Create an array of fruits and access specific elements.

**Prompt:**

```
Create a JavaScript array named `fruits` containing the strings "apple", "banana", and "cherry". Then, write code to print the second fruit in the array to the console.
```

**AI-Generated Code (Expected):**

```
const fruits = ["apple", "banana", "cherry"];
console.log(fruits[1]); // Expected output: banana
```

- **Analysis:** The AI correctly declares an array and uses bracket notation ( `[ 1 ]` ) to access the element at index 1 (the second element).

## Example 2: Adding and Removing Elements

**Goal:** Add a new fruit to the end of the array and remove the last fruit.

**Prompt:**

Given the `fruits` array: `["apple", "banana", "cherry"]`.

First, add "grape" to the end of the array.

Second, remove the last element from the array.

Finally, print the modified array to the console.

**AI-Generated Code (Expected):**

```
const fruits = ["apple", "banana", "cherry"];

fruits.push("grape"); // Adds "grape" to the end
console.log(fruits); // Expected: ["apple", "banana", "cherry", "grape"]

fruits.pop(); // Removes the last element ("grape")
console.log(fruits); // Expected: ["apple", "banana", "cherry"]
```

- **Analysis:** The AI correctly uses `push()` to add an element to the end and `pop()` to remove the last element, demonstrating standard array manipulation methods.

## Example 3: Iterating Over Arrays

**Goal:** Print each fruit in the array using a loop.

**Prompt:**

Given the `fruits` array: `["apple", "banana", "cherry"]`.

Write JavaScript code to iterate over this array and print each fruit to the console, one per line.

**AI-Generated Code (Expected):**

```
const fruits = ["apple", "banana", "cherry"];

for (let i = 0; i < fruits.length; i++) {
 console.log(fruits[i]);
}

// Or using forEach (more modern):
// fruits.forEach(function(fruit) {
// console.log(fruit);
// });
```

- **Analysis:** The AI provides both a traditional `for` loop and the more modern `forEach` method, showing different ways to process each item in an array.

## Objects: Key-Value Collections

---

Objects are used to store collections of data and more complex entities. Data is stored as key-value pairs, where keys are strings (or Symbols) and values can be any data type.

### Example 4: Creating and Accessing Objects

**Goal:** Create an object representing a person and access their properties.

**Prompt:**

```
Create a JavaScript object named `person` with the following properties:
- `firstName`: "John"
- `lastName`: "Doe"
- `age`: 30
- `isStudent`: false

Then, print the person's `firstName` and `age` to the console.
```

**AI-Generated Code (Expected):**

```
const person = {
 firstName: "John",
 lastName: "Doe",
 age: 30,
 isStudent: false
};

console.log(person.firstName); // Expected output: John
console.log(person.age); // Expected output: 30
```

- **Analysis:** The AI correctly defines an object using curly braces and accesses properties using dot notation ( `person.firstName` ).

## Example 5: Modifying Object Properties

**Goal:** Change a person's age and add a new property.

**Prompt:**

Given the `person` object: `{ firstName: "John", lastName: "Doe", age: 30, isStudent: false }`.

First, change the `age` property to 31.

Second, add a new property `city` with the value "New York".

Finally, print the updated `person` object to the console.

**AI-Generated Code (Expected):**

```
const person = {
 firstName: "John",
 lastName: "Doe",
 age: 30,
 isStudent: false
};

person.age = 31; // Modifying an existing property
person.city = "New York"; // Adding a new property

console.log(person);
// Expected: { firstName: "John", lastName: "Doe", age: 31,
isStudent: false, city: "New York" }
```

- **Analysis:** The AI demonstrates how to update existing properties and add new ones to an object.

## Combining Arrays and Objects

---

Real-world data often involves collections of objects, such as a list of users or products. Arrays of objects are very common.

### Example 6: Array of Objects

**Goal:** Create an array of user objects and find a specific user.

**Prompt:**

Create a JavaScript array named `users`. Each element in the array should be an object with `id`, `name`, and `email` properties.

Add two user objects:

- User 1: id 1, name "Alice", email "alice@example.com"
- User 2: id 2, name "Bob", email "bob@example.com"

Then, write a function `findUserId` that takes an `id` and the `users` array, and returns the user object with that `id`. If no user is found, return `null`.

### AI-Generated Code (Expected):

```
const users = [
 { id: 1, name: "Alice", email: "alice@example.com" },
 { id: 2, name: "Bob", email: "bob@example.com" }
];

function findUserId(id, userList) {
 for (let i = 0; i < userList.length; i++) {
 if (userList[i].id === id) {
 return userList[i];
 }
 }
 return null;
}

// Example usage:
// console.log(findUserId(1, users)); // Expected: { id: 1, name:
// "Alice", email: "alice@example.com" }
// console.log(findUserId(3, users)); // Expected: null
```

- **Analysis:** This example showcases how to create and iterate over an array of objects, a pervasive pattern in web development. The `findUserId` function demonstrates searching within this structure.

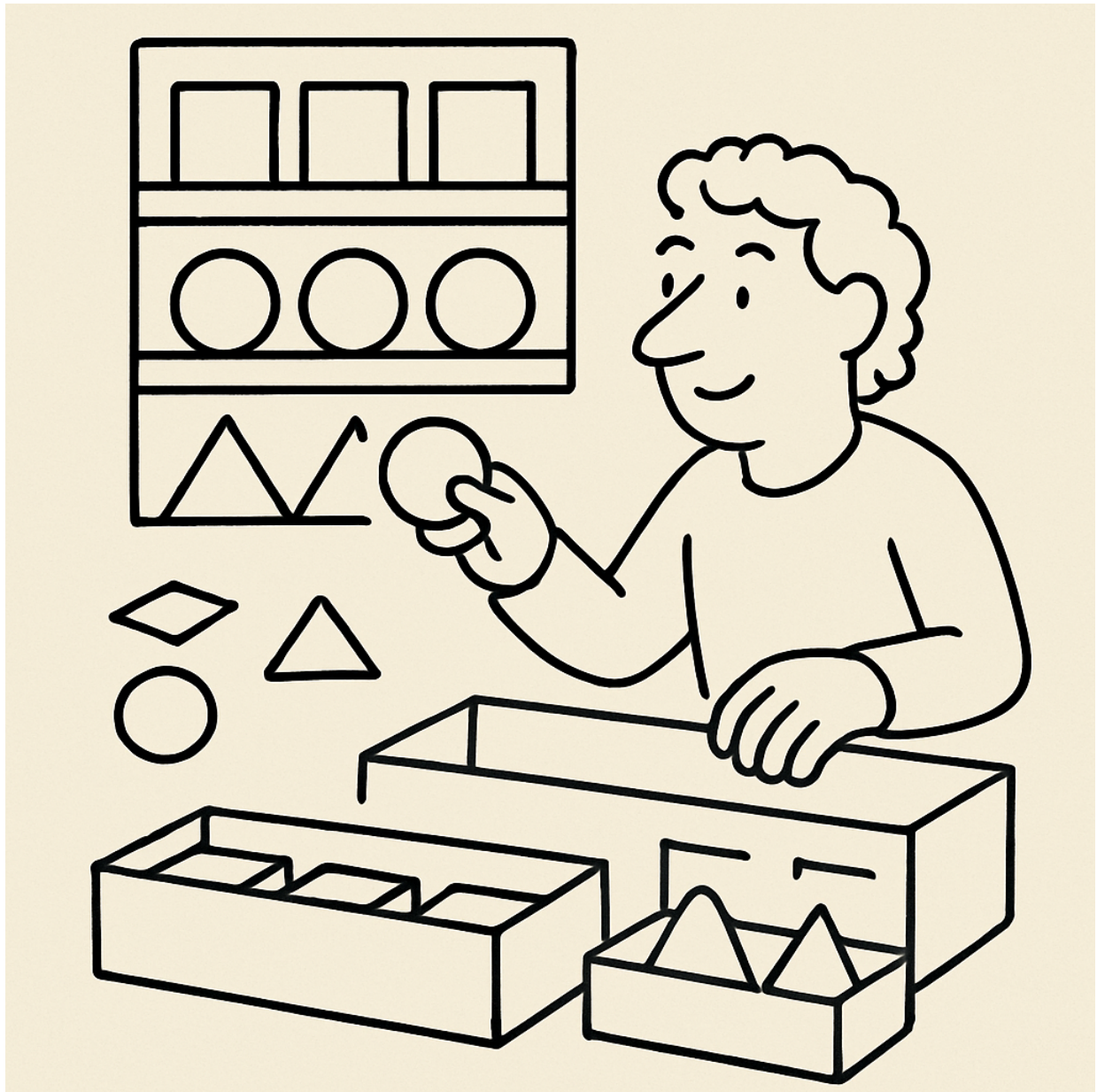
## Conclusion: Structuring Your Data

Arrays and objects are the fundamental building blocks for structuring data in JavaScript. By mastering how to prompt AI to create, manipulate, and query these structures, you gain the ability to handle virtually any kind of information your applications might need. In the next chapter, we will learn how to control the flow of your programs using conditional statements and loops.

---

### References:

- [MDN Web Docs: Arrays](#)
- [MDN Web Docs: Objects](#)
- [JavaScript.info: Objects](#)





# Chapter 7: Controlling the Flow - Logic, Conditionals, and Loops

## Guiding Your Code: Decisions and Repetition

---

In programming, it's often necessary to make decisions based on certain conditions or to repeat a set of actions multiple times. Just as Forth uses conditional words and looping constructs to control program execution, JavaScript provides powerful mechanisms for **controlling the flow** of your code. Understanding these concepts and how to prompt AI to generate them is fundamental to building dynamic and interactive applications.

This chapter will cover conditional statements ( `if` , `else if` , `else` , `switch` ) and looping constructs ( `for` , `while` , `do...while` , `forEach` ), demonstrating how to use AI to implement logic and repetition in your JavaScript code.

## Conditional Statements: Making Decisions

---

Conditional statements allow your program to execute different blocks of code based on whether a specified condition is true or false.

### The `if` , `else if` , and `else` Statements

These are the most common conditional statements, allowing for branching logic.

#### Prompt:

```
Write a JavaScript function `checkNumber` that takes one argument, `num`
- If `num` is positive, print "The number is positive."
- If `num` is negative, print "The number is negative."
- If `num` is zero, print "The number is zero."
```

#### AI-Generated Code (Expected):

```
function checkNumber(num) {
 if (num > 0) {
 console.log("The number is positive.");
 } else if (num < 0) {
 console.log("The number is negative.");
 } else {
 console.log("The number is zero.");
 }
}

// Example usage:
// checkNumber(5); // Output: The number is positive.
// checkNumber(-3); // Output: The number is negative.
// checkNumber(0); // Output: The number is zero.
```

- **Analysis:** The AI correctly uses `if`, `else if`, and `else` to handle three different conditions, demonstrating clear branching logic.

## The `switch` Statement

The `switch` statement is an alternative to `if...else if...else` when you have many possible execution paths based on a single variable's value.

### Prompt:

```
Write a JavaScript function `getWeekdayName` that takes a number `day`
(0 for Sunday, 1 for Monday, etc.) and returns the corresponding weekday
name. Use a `switch` statement. If the number is not between 0 and 6,
return "Invalid day."
```

### AI-Generated Code (Expected):

```
function getWeekdayName(day) {
 switch (day) {
 case 0:
 return "Sunday";
 case 1:
 return "Monday";
 case 2:
 return "Tuesday";
 case 3:
 return "Wednesday";
 case 4:
 return "Thursday";
 case 5:
 return "Friday";
 case 6:
 return "Saturday";
 default:
 return "Invalid day.";
 }
}

// Example usage:
// console.log(getWeekdayName(3)); // Output: Wednesday
// console.log(getWeekdayName(7)); // Output: Invalid day.
```

- **Analysis:** The AI correctly implements the `switch` statement, including the `default` case for invalid input, which is good practice.

## Looping Constructs: Repeating Actions

---

Loops allow you to execute a block of code repeatedly until a specific condition is met. This is essential for processing lists of data, performing calculations, or generating repetitive content.

### The `for` Loop

The `for` loop is commonly used when you know exactly how many times you want to repeat a block of code.

#### Prompt:

```
Write a JavaScript `for` loop that prints numbers from 1 to 5 to the console.
```

#### AI-Generated Code (Expected):

```
for (let i = 1; i <= 5; i++) {
 console.log(i);
}

// Expected output:
// 1
// 2
// 3
// 4
// 5
```

- **Analysis:** The AI generates a standard `for` loop, correctly setting the initialisation, condition, and increment.

## The `while` Loop

The `while` loop executes a block of code as long as a specified condition is true. It's often used when the number of iterations is not known beforehand.

### Prompt:

```
Write a JavaScript `while` loop that starts with a variable `count`
initialised to 0. The loop should continue as long as `count` is less
than 3. Inside the loop, print the current value of `count` and then
increment `count` by 1.
```

### AI-Generated Code (Expected):

```
let count = 0;
while (count < 3) {
 console.log(count);
 count++;
}

// Expected output:
// 0
// 1
// 2
```

- **Analysis:** The AI correctly sets up a `while` loop, ensuring the `count` variable is incremented to avoid an infinite loop.

## The `do...while` Loop

The `do...while` loop is similar to `while`, but it guarantees that the loop body will

execute at least once, because the condition is evaluated *after* the first iteration.

**Prompt:**

```
Write a JavaScript `do...while` loop that prints "Hello!" to the console once, even if a condition `isDone` is initially true.
```

**AI-Generated Code (Expected):**

```
let isDone = true;

do {
 console.log("Hello!");
} while (!isDone);

// Expected output:
// Hello!
```

- **Analysis:** The AI demonstrates the `do...while` loop's characteristic of executing at least once, regardless of the initial condition.

## The `for...of` Loop (for Iterables)

Introduced in ES6, the `for...of` loop offers a more straightforward way to iterate over iterable objects, including arrays, strings, Maps, Sets, and more.

**Prompt:**

```
Given an array of names: `const names = ["Alice", "Bob", "Charlie"];`

Write a JavaScript `for...of` loop to print each name to the console.
```

**AI-Generated Code (Expected):**

```
const names = ["Alice", "Bob", "Charlie"];

for (const name of names) {
 console.log(name);
}

// Expected output:
// Alice
// Bob
// Charlie
```

- **Analysis:** The AI correctly uses the `for...of` loop, which is often preferred for its readability when iterating over array elements.

## Conclusion: Directing Your Program's Path

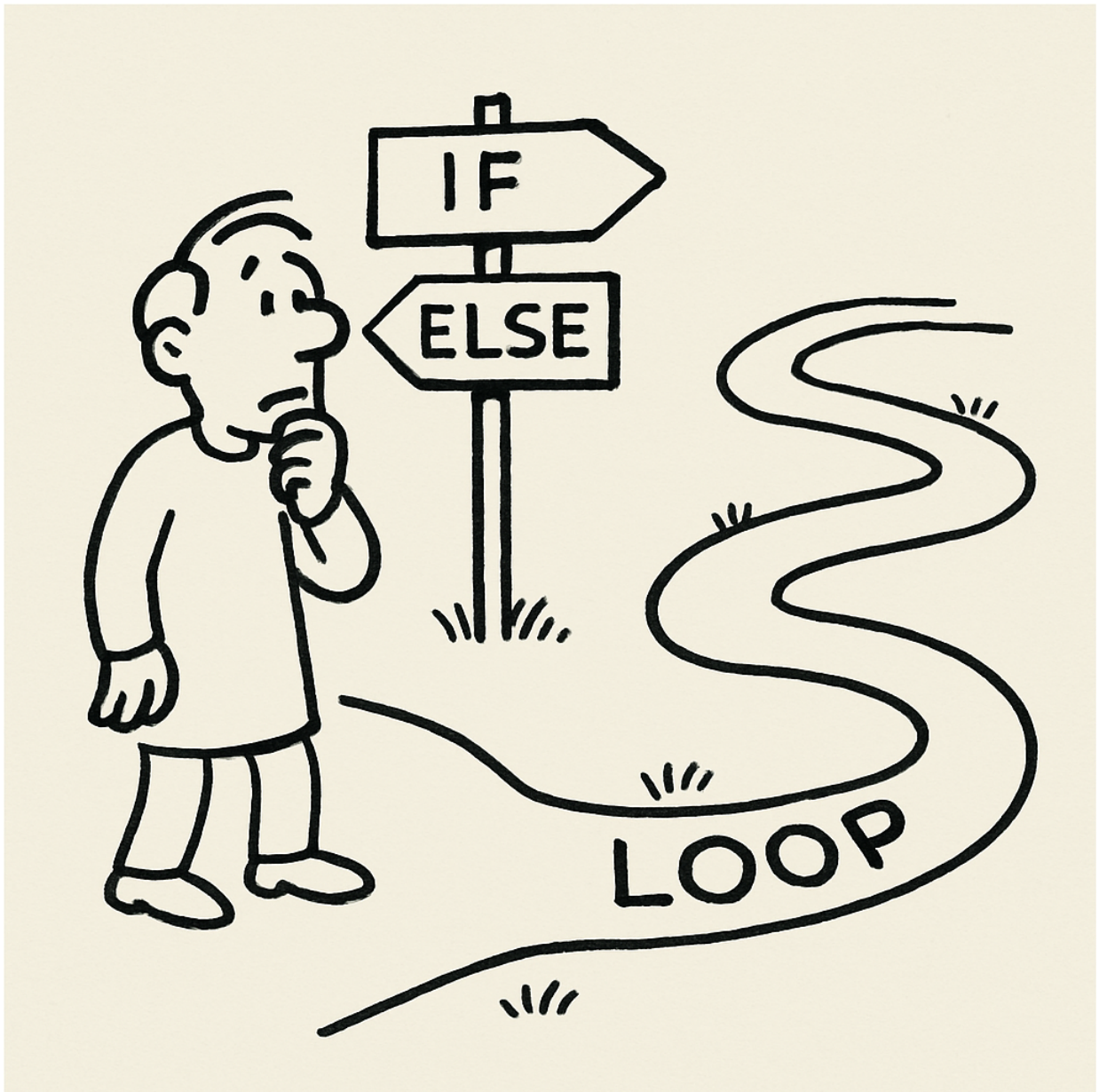
---

Conditional statements and loops are the bedrock of program logic. They enable your JavaScript code to respond dynamically to various situations and process data efficiently. By effectively prompting AI to generate these control flow structures, you gain immense power in directing your program's behaviour. In the next chapter, we will delve into the art of refining and debugging AI-generated code.



### References:

- [MDN Web Docs: Conditional \(ternary\) operator](#)
- [MDN Web Docs: if...else](#)
- [MDN Web Docs: switch](#)
- [MDN Web Docs: for loop](#)
- [MDN Web Docs: while loop](#)
- [MDN Web Docs: do...while](#)
- [MDN Web Docs: for...of](#)





# Chapter 8: The Art of Refinement - Debugging and Improving AI-Generated Code

## From Raw Output to Polished Product: The Developer's Touch

---

AI-generated code, while impressive, is rarely perfect. Just as a sculptor refines a raw block of marble into a masterpiece, a developer must refine the AI's output into robust, efficient, and maintainable code. This process involves **debugging** (identifying and correcting errors) and **improving** (optimising and refining) the generated code. This chapter will equip you with the skills to critically evaluate AI-generated code and transform it into production-ready solutions.

## Understanding Common Issues with AI-Generated Code

---

Before we dive into refinement, it's important to recognise the typical shortcomings of AI-generated code:

1. **Syntax Errors:** While AIs are generally good with syntax, minor typos or incorrect punctuation can still occur, especially in complex snippets or less common language features.
2. **Logical Errors:** The code may be syntactically correct, but it doesn't do what you intended. This often stems from ambiguous prompts or the AI misinterpreting your requirements.
3. **Inefficiency:** The generated code may work, but it may not be the most performant or resource-efficient solution, especially for large datasets or high-traffic applications.
4. **Lack of Best Practices:** AI might not always adhere to the latest coding standards, security practices, or idiomatic patterns for a given language or framework.
5. **Incomplete or Generic Solutions:** The AI may provide a general solution when a specific, nuanced one is required, or it may omit crucial parts of the code.
6. **Security Vulnerabilities:** Poorly prompted or unreviewed AI code can introduce

security flaws.

7. **Hallucinations:** The AI might confidently generate code that looks plausible but is entirely incorrect or nonsensical in context.

## Debugging AI-Generated Code

---

Debugging is the process of identifying and removing errors from computer hardware or software. When working with AI-generated code, your debugging process will often involve:

### 1. Read and Understand the Code

Before running any code, take the time to review the AI-generated code. Does it make sense? Does it align with your understanding of the problem? Look for obvious logical flaws or missing pieces.

### 2. Use Your Development Environment's Tools

Your VS Code workbench is equipped with powerful debugging tools.

- **Console Logging ( `console.log()` ):** This is your most basic and often most effective debugging tool. Sprinkle `console.log()` statements throughout your code to inspect variable values at different points in execution.

**Prompt:**

Add `console.log` statements to this JavaScript function to show the value of 'sum' after each iteration of the loop.

```
function calculateSum(numbers) {
 let sum = 0;
 for (let i = 0; i < numbers.length; i++) {
 sum += numbers[i];
 }
 return sum;
}
```

**AI-Generated Code (Expected):**

```
function calculateSum(numbers) {
 let sum = 0;
 for (let i = 0; i < numbers.length; i++) {
 sum += numbers[i];
 console.log(`Sum after iteration ${i}: ${sum}`);
 // Added console.log
 }
 return sum;
}
```

- **Debugger:** VS Code has a built-in debugger that allows you to pause code execution, step through lines, inspect variables, and set breakpoints. This is invaluable for understanding complex logic.

#### **Prompt:**

Explain how to set a breakpoint in Visual Studio Code for a JavaScript file and step through the code line by line.

#### **AI-Generated Explanation (Expected):**

To set a breakpoint in VS Code:

1. Open your JavaScript file in the editor.
2. Click in the gutter (the space to the left of the line numbers) next to the line of code where you want execution to pause. A red dot will appear, indicating a breakpoint.

To step through code:

1. Go to the 'Run and Debug' view (Ctrl+Shift+D or Cmd+Shift+D).
2. Click the 'Run and Debug' button (green play icon).
3. When execution pauses at your breakpoint, use the debugging controls in the top bar:
  - 'Step Over' (F10): Executes the current line and moves to the next.
  - 'Step Into' (F11): If the current line calls a function, it steps into that function.
  - 'Step Out' (Shift+F11): Steps out of the current function.
  - 'Continue' (F5): Continues execution until the next breakpoint or end of the program.

### **3. Test Cases**

Always test AI-generated code with various inputs, including edge cases (e.g., empty arrays, zero values, invalid inputs). This helps uncover logical errors.

# Improving AI-Generated Code: Beyond Correctness

---

Once the code is functional, the next step is to improve it. This involves optimisation, refactoring, and adhering to best practices.

## 1. Refactoring for Readability and Maintainability

Refactoring refers to the process of restructuring existing computer code without altering its external behaviour. It makes the code cleaner, easier to understand, and simpler to maintain.

### Prompt:

Refactor this JavaScript function to improve readability and use more modern array methods. The function currently filters an array of numbers to keep only even numbers.

```
function filterEvenNumbers(numbers) {
 let evenNumbers = [];
 for (let i = 0; i < numbers.length; i++) {
 if (numbers[i] % 2 === 0) {
 evenNumbers.push(numbers[i]);
 }
 }
 return evenNumbers;
}
```

### AI-Generated Code (Expected):

```
function filterEvenNumbers(numbers) {
 return numbers.filter(num => num % 2 === 0);
}
```

- **Analysis:** The AI utilises the `filter()` array method, which is a more concise and readable approach to achieving the same result as the `for` loop, demonstrating modern JavaScript practices.

## 2. Optimising for Performance

Sometimes, AI might generate code that works but isn't the most efficient. You can prompt it to optimise.

### Prompt:

Optimise this JavaScript function for improved performance when handling extremely large arrays. The function finds the sum of all numbers in an array.

```
function sumArray(arr) {
 let total = 0;
 for (const num of arr) {
 total += num;
 }
 return total;
}
```

#### AI-Generated Code (Expected):

```
function sumArray(arr) {
 // For very large arrays, a simple for loop is often the most
 performant
 // as it avoids the overhead of higher-order functions like reduce.
 let total = 0;
 for (let i = 0; i < arr.length; i++) {
 total += arr[i];
 }
 return total;
}

// Alternative (often less performant for very large arrays, but more
// concise):
// function sumArray(arr) {
// return arr.reduce((acc, curr) => acc + curr, 0);
// }
```

- **Analysis:** The AI correctly identifies that for very large arrays, a traditional `for` loop can be more performant than `reduce` due to less overhead, and explains why. This shows an understanding of performance considerations.

### 3. Adding Error Handling and Robustness

Production-ready code needs to handle unexpected inputs or situations gracefully.

**Prompt:**

Add error handling to this JavaScript function. It should throw an error if the input 'data' is not an array.

```
function processData(data) {
 // Assume data is an array for now,
 return data.map(item => item * 2);
}
```

### AI-Generated Code (Expected):

```
function processData(data) {
 if (!Array.isArray(data)) {
 throw new Error("Input must be an array.");
 }
 return data.map(item => item * 2);
}
```

- **Analysis:** The AI adds a check using `Array.isArray()` and throws a meaningful error, making the function more robust.

## Conclusion: The Human in the Loop

---

Debugging and improving AI-generated code are critical skills that elevate you from a mere prompt-giver to an actual developer. The AI is a powerful assistant, but your expertise in identifying issues, understanding best practices, and applying refinement techniques is what transforms raw output into high-quality software. Embrace this iterative process, and you'll find yourself building more reliable and efficient applications. In the next chapter, we will explore how to create larger, more structured applications using classes, modules, and APIs.

---

### References:

- [MDN Web Docs: Debugging JavaScript](#)
- [Visual Studio Code: Debugging](#)
- [Refactoring Guru](#)
- [MDN Web Docs: Array.prototype.filter\(\)](#)





# Chapter 9: Building Larger Structures - Classes, Modules, and APIs

## Beyond Simple Functions: Organising Complex Code

---

As your JavaScript applications grow in complexity, simply writing functions and variables in a single file becomes unwieldy. Just as Forth allows you to build complex programs from smaller, well-defined "words," JavaScript provides powerful features for organising your code into reusable and maintainable structures. This chapter will introduce you to **Classes**, **Modules**, and **APIs**, and how to leverage AI to work with them effectively.

### Classes: Blueprints for Objects

---

In object-oriented programming, a **class** is a blueprint for creating objects (a particular instance of the class). It encapsulates data (properties) and functions (methods) that operate on that data. While JavaScript is primarily a prototype-based language, it offers a `class` syntax that makes object-oriented programming more familiar to developers from other languages.

#### Example 1: Defining a Simple Class

**Goal:** Define a `Car` class with properties for `make`, `model`, and `year`, and a method to display car information.

**Prompt:**

```
Define a JavaScript class named `Car`. The constructor should take `make`
and `year` as arguments and assign them as properties.
Add a method `displayInfo` that returns a string like
"This is a [year] [make] [model]."
```

**AI-Generated Code (Expected):**

```
class Car {
 constructor(make, model, year) {
 this.make = make;
 this.model = model;
 this.year = year;
 }

 displayInfo() {
 return `This is a ${this.year} ${this.make} ${this.model}.`;
 }
}

// Example usage:
// const myCar = new Car("Toyota", "Camry", 2020);
// console.log(myCar.displayInfo()); // Expected: This is a 2020 Toyota Camry
```

- **Analysis:** The AI correctly defines a class with a constructor and a method, demonstrating the basic structure of a JavaScript class.

## Example 2: Class Inheritance

Classes can inherit properties and methods from other classes, promoting code reuse.

**Goal:** Create an `ElectricCar` class that inherits from `Car` and adds a `batteryLife` property.

**Prompt:**

```
Create a JavaScript class named `ElectricCar` that extends the `Car` class.
Its constructor should take `make`, `model`, `year`, and `batteryLife` as arguments.
The `displayInfo` method should be overridden to also include battery life information.
```

**AI-Generated Code (Expected):**

```
class ElectricCar extends Car {
 constructor(make, model, year, batteryLife) {
 super(make, model, year); // Call the parent Car constructor
 this.batteryLife = batteryLife;
 }

 displayInfo() {
 return `${super.displayInfo()} It has a battery life of ${this.batteryLife} kWh.`;
 }
}

// Example usage:
// const tesla = new ElectricCar("Tesla", "Model 3", 2023, 75);
// console.log(tesla.displayInfo()); // Expected: This is a 2023 Tesla Model 3. It has a battery life of 75 kWh.
```

- **Analysis:** The AI demonstrates class inheritance using `extends` and `super()`, as well as method overriding, which are key concepts in object-oriented design.

## Modules: Organising Your Codebase

As your projects grow, you'll want to split your code into separate files (modules) to keep it organised and prevent naming conflicts. JavaScript modules ( `import` and `export` ) provide a standardised way to do this.

### Example 3: Exporting and Importing Functions

**Goal:** Create a module that exports a utility function and imports it into another file.

**Prompt:**

Create two JavaScript files.

File 1 (utils.js): Export a function named `capitalise` that takes a string and returns the string with the first letter capitalised.

File 2 (main.js): Import the `capitalise` function from `utils.js` and use it to capitalise the string "hello world", then print the result to the console.

**AI-Generated Code (Expected):**

`utils.js` :

```
export function capitalize(str) {
 if (!str) return "";
 return str.charAt(0).toUpperCase() + str.slice(1);
}
```

**main.js :**

```
import { capitalize } from './utils.js';

const message = "hello world";
const capitalizedMessage = capitalize(message);
console.log(capitalizedMessage); // Expected: Hello world
```

- **Analysis:** The AI correctly uses `export` in `utils.js` and `import` in `main.js`, demonstrating how to modularise code. It also includes basic input validation in the `capitalise` function.

## APIs: Interacting with External Services

---

An **API (Application Programming Interface)** is a set of rules and definitions that allow different software applications to communicate with each other. In web development, you'll frequently interact with web APIs to fetch data from servers (e.g., weather data, stock prices, social media feeds) or send data to them.

JavaScript provides the `fetch` API for making network requests.

### Example 4: Fetching Data from a Public API

**Goal:** Fetch a random joke from a public API and display it.

**Prompt:**

```
Write a JavaScript function `fetchRandomJoke` that fetches a random joke
(https://v2.jokeapi.dev/joke/Any?safe-mode). The function should return
(either `joke` or `setup` + `delivery`).
Handle potential errors during the fetch operation.
```

**AI-Generated Code (Expected):**

```

async function fetchRandomJoke() {
 try {
 const response = await fetch("https://v2.jokeapi.dev/joke/Any?safe-mo
 if (!response.ok) {
 throw new Error(`HTTP error! status: ${response.status}`);
 }
 const data = await response.json();

 if (data.type === "single") {
 return data.joke;
 } else if (data.type === "twopart") {
 return `${data.setup} ... ${data.delivery}`;
 }
 } catch (error) {
 console.error("Could not fetch joke:", error);
 return "Failed to fetch a joke. Please try again later.";
 }
}

// Example usage (in an async context):
// (async () => {
// const joke = await fetchRandomJoke();
// console.log(joke);
// })();

```

- **Analysis:** The AI correctly uses the `fetch` API with `async/await` for asynchronous operations, parses JSON data, and handles different joke types (single vs. two-part). Crucially, it includes error handling, which is vital for robust API interactions.

## Conclusion: Building Scalable Applications

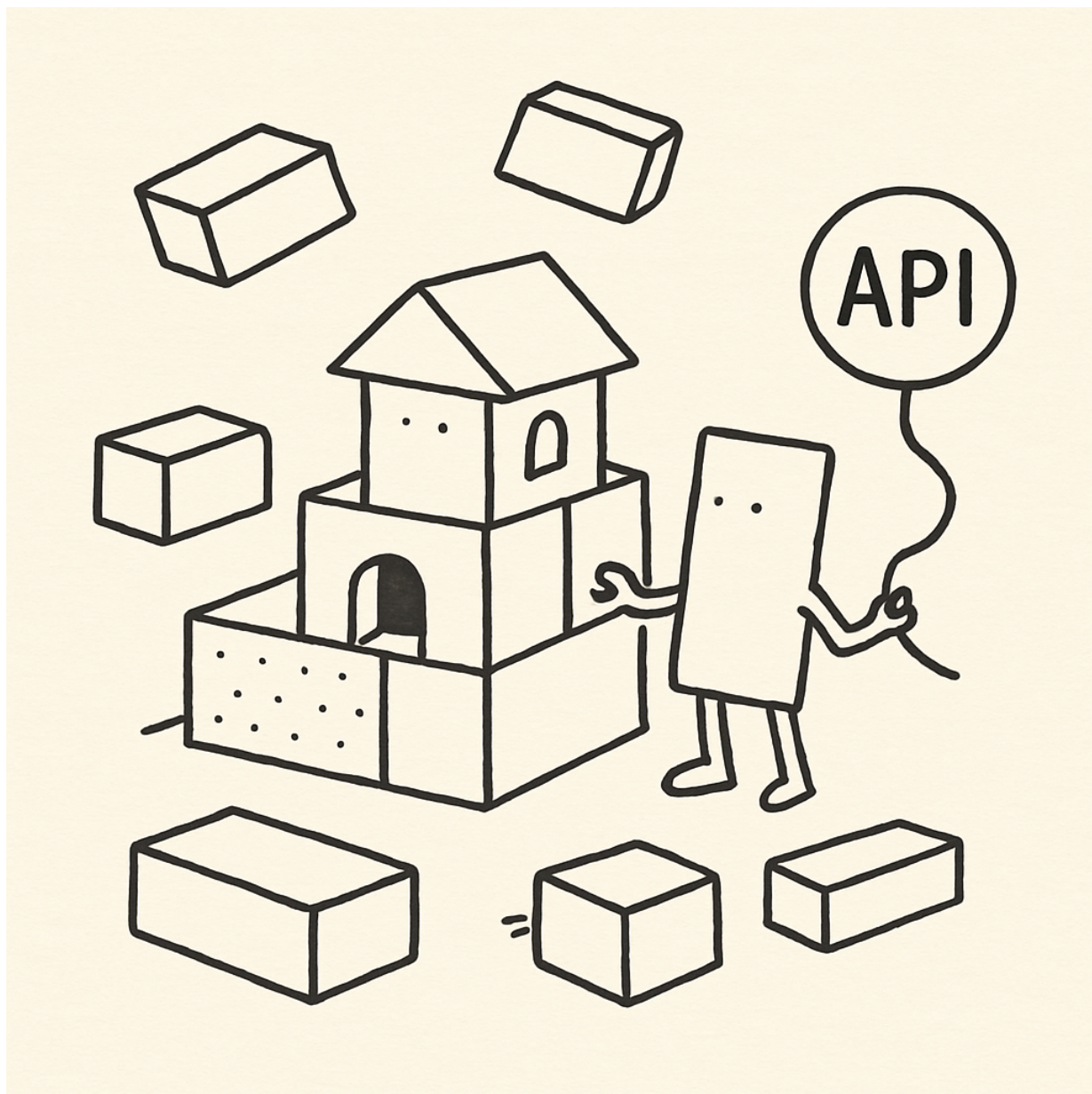
---

Classes, Modules, and APIs are essential tools for building larger, more organised, and interconnected JavaScript applications. By mastering how to prompt AI to generate and work with these structures, you can tackle more ambitious projects and integrate your code with external services. In the next chapter, we will bring all these concepts together to build a complete, real-world application.

---

### References:

- [MDN Web Docs: Classes](#)
- [MDN Web Docs: JavaScript modules](#)
- [MDN Web Docs: Using the Fetch API](#)
- [JokeAPI Documentation](#)



# Chapter 10: Bringing Code to Life - Interactive Web Development

## The Dynamic Web: Interacting with the User

---

Forth is known for its interactive nature, allowing you to build and test code incrementally. Similarly, a significant part of JavaScript's power lies in its ability to make web pages dynamic and interactive. This involves responding to user actions (such as clicks or key presses) and modifying the content and style of a web page after it has loaded. This chapter will introduce you to the **Document Object Model (DOM)**, **event handling**, and how to prompt AI to create interactive web experiences.

## The Document Object Model (DOM): Your Web Page's Structure

---

The DOM is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as a tree of nodes and objects, allowing programming languages to interact with the page.

Think of the DOM as a tree-like structure where each HTML element (like `<div>`, `<p>`, `<img>`) is a node. JavaScript can access and modify these nodes.

### Example 1: Accessing and Modifying Elements

**Goal:** Change the text content of an HTML element.

**Prompt:**

```
I have an HTML paragraph with the ID "myParagraph". Write JavaScript code to change its content to "Hello from JavaScript!".
```

**AI-Generated Code (Expected):**



```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <title>DOM Example</title>
</head>
<body>
 <p id="myParagraph">Original text.</p>

 <script>
 const paragraphElement = document.getElementById("myParagraph");
 paragraphElement.textContent = "Hello from JavaScript!";
 </script>
</body>
</html>
```

- **Analysis:** The AI correctly uses `document.getElementById()` to select the element and `textContent` to change its content. This is a fundamental DOM manipulation technique.

## Example 2: Changing Element Styles

**Goal:** Change the background colour of a div when a button is clicked.

**Prompt:**

```
I have an HTML div with the ID "myBox" and a button with the ID "changeColor".
Write JavaScript code so that when the button is clicked, the background color
changes to "lightblue".
```

**AI-Generated Code (Expected):**

```

<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <title>Style Change Example</title>
 <style>
 #myBox {
 width: 100px;
 height: 100px;
 background-color: lightgray;
 border: 1px solid black;
 }
 </style>
</head>
<body>
 <div id="myBox"></div>
 <button id="changeColorButton">Change Color</button>

 <script>
 const myBox = document.getElementById("myBox");
 const changeColorButton = document.getElementById("changeColorButton");

 changeColorButton.addEventListener("click", function() {
 myBox.style.backgroundColor = "lightblue";
 });
 </script>
</body>
</html>

```

- **Analysis:** This example introduces event handling ( `addEventListener` ) and directly manipulating an element's style using `element.style.property` .

## Event Handling: Responding to User Actions

---

Events are actions or occurrences that happen in the system you are programming, which the system tells you about so you can respond to them. Common web events include clicks, key presses, form submissions, and page loads.

### Example 3: Click Events

**Goal:** Display an alert message when a button is clicked.

**Prompt:**

Write JavaScript code that, when a button with the ID "showAlertButton" is clicked, shows an alert box with the message "Button Clicked!".

### AI-Generated Code (Expected):

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <title>Click Event Example</title>
</head>
<body>
 <button id="showAlertButton">Click Me</button>

 <script>
 const showAlertButton = document.getElementById("showAlertButton");

 showAlertButton.addEventListener("click", function() {
 alert("Button Clicked!");
 });
 </script>
</body>
</html>
```

- **Analysis:** This demonstrates the basic `addEventListener` pattern for handling click events.

## Example 4: Input Events and Form Handling

**Goal:** Get text from an input field and display it in a paragraph when a form is submitted.

### Prompt:

I have an HTML form with an input field (id="nameInput") and a submit button (id="submit"). There's also a paragraph (id="displayArea").

Write JavaScript code to:

- Prevent the default form submission behaviour.
- When the form is submitted, get the value from "nameInput".
- Display the input value in the "displayArea" paragraph.

### AI-Generated Code (Expected):

```

<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <title>Form Example</title>
</head>
<body>
 <form id="myForm">
 <label for="nameInput">Enter your name:</label>
 <input type="text" id="nameInput">
 <button type="submit">Submit</button>
 </form>
 <p id="displayArea"></p>

 <script>
 const myForm = document.getElementById("myForm");
 const nameInput = document.getElementById("nameInput");
 const displayArea = document.getElementById("displayArea");

 myForm.addEventListener("submit", function(event) {
 event.preventDefault(); // Prevent default form submission
 const name = nameInput.value;
 displayArea.textContent = `Hello, ${name}!`;
 });
 </script>
</body>
</html>

```

- **Analysis:** This example shows how to handle form submissions, prevent the default browser behaviour ( `event.preventDefault()` ), and retrieve input values from form fields.

## Conclusion: Making Your Web Pages Alive

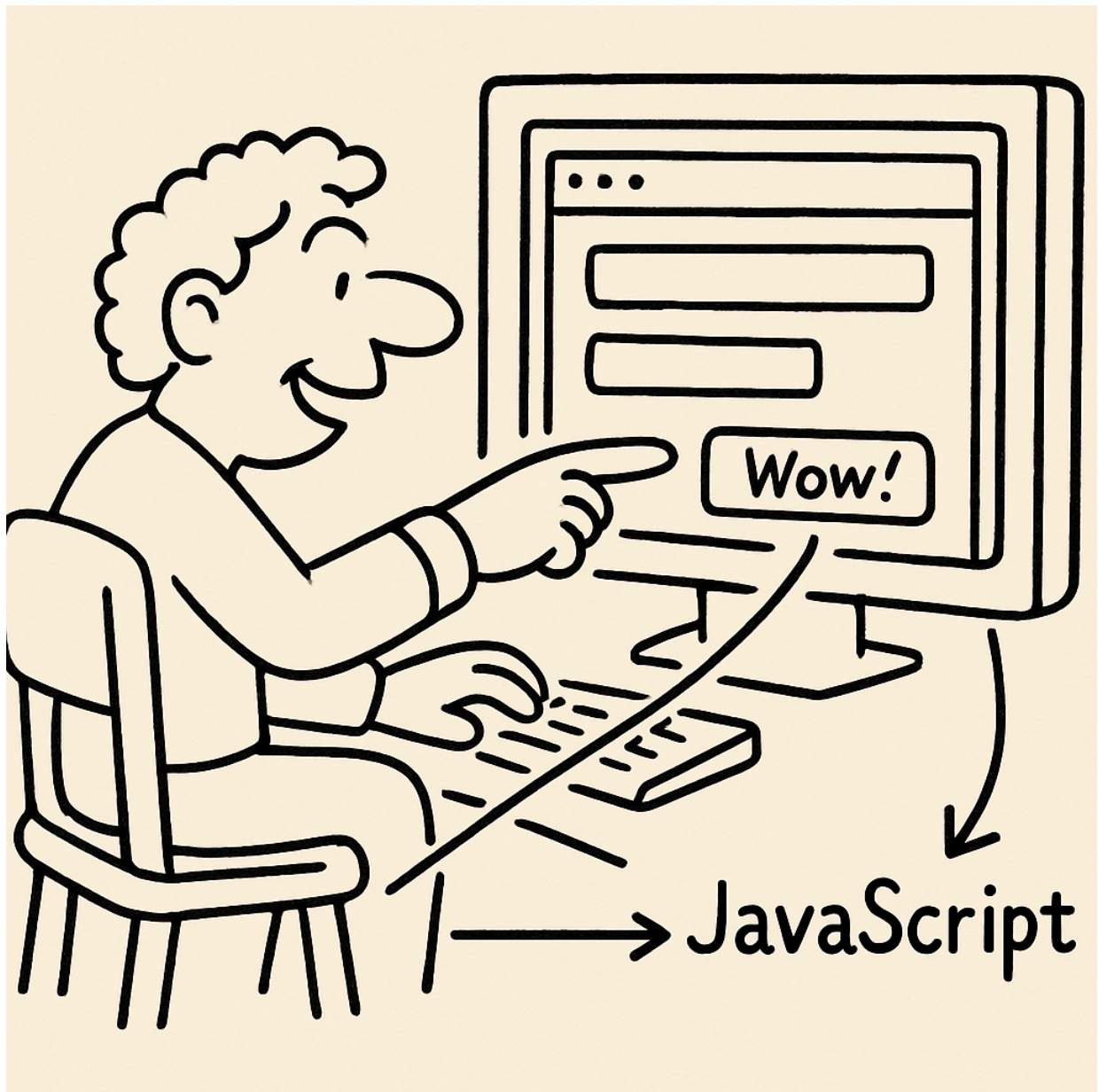
---

By understanding the DOM and mastering event handling, you can transform static web pages into dynamic, interactive experiences. Prompting AI to assist with these tasks will significantly accelerate your development process, enabling you to focus on the user experience rather than the intricate details of DOM manipulation. In the next chapter, we will explore more advanced prompting techniques to get even better results from your AI assistant.

---

### References:

- [MDN Web Docs: Introduction to the DOM](#)
- [MDN Web Docs: EventTarget.addEventListener\(\)](#)
- [MDN Web Docs: HTMLFormElement.submit\(\)](#)



# Chapter 11: Advanced Prompting Techniques - Getting More from Your AI Assistant

## Beyond the Basics: Mastering the Art of Conversation with AI

---

In previous chapters, you've learned the fundamentals of prompt engineering and how to get AI to generate basic JavaScript code, handle data structures, control program flow, and even create interactive web elements. Now, it's time to elevate your prompting skills. An advanced prompt engineer knows how to craft sophisticated instructions to unlock the full potential of AI code generation.

This chapter will delve into more advanced prompting techniques, including:

- **Few-Shot Prompting:** Providing examples to guide the AI.
- **Chain-of-Thought Prompting:** Encouraging the AI to reason step-by-step.
- **Role-Playing and Persona:** Guiding the AI's perspective.
- **Iterative Prompting and Refinement:** The conversational approach to code generation.
- **Constraint-Based Prompting:** Imposing specific rules and limitations.

## Few-Shot Prompting: Learning by Example

---

Few-shot prompting involves giving the AI a few examples of input-output pairs before asking it to complete a new task. This helps the AI understand the desired format, style, and logic, especially for tasks that are nuanced or require a specific pattern.

### Example 1: Formatting Data

**Goal:** Convert an array of objects into a specific string format.

**Prompt:**

Convert the following array of user objects into a comma-separated string formatted as "Name (Email)".

Example:

Input: [

```
{ name: "Alice", email: "alice@example.com" },
{ name: "Bob", email: "bob@example.com" }
```

]

Output: "Alice (alice@example.com), Bob (bob@example.com)"

Now, convert this input:

Input: [

```
{ name: "Charlie", email: "charlie@example.com" },
{ name: "David", email: "david@example.com" },
{ name: "Eve", email: "eve@example.com" }
```

]

### AI-Generated Code (Expected):

```
const users = [
 { name: "Charlie", email: "charlie@example.com" },
 { name: "David", email: "david@example.com" },
 { name: "Eve", email: "eve@example.com" }
];

const formattedString = users.map(user => `${user.name} (${user.email})`)
console.log(formattedString);
// Expected Output: "Charlie (charlie@example.com), David (david@example.com), Eve (eve@example.com)"
```

- **Analysis:** By providing a clear example, the AI understands the exact output format required, even for a slightly complex transformation.

## Chain-of-Thought Prompting: Thinking Step-by-Step

---

Chain-of-thought prompting encourages the AI to break down a complex problem into intermediate steps, showing its reasoning process. This can lead to more accurate and reliable results, especially for multi-step coding tasks.

### Example 2: Complex Logic with Explanation

**Goal:** Generate a function to calculate Fibonacci numbers, explaining the logic.

## Prompt:

Let's think step by step. First, define what a Fibonacci sequence is. Second, write a JavaScript function `fibonacci` that takes a number `n` and returns the nth Fibonacci number. Third, explain the logic of your function.

## AI-Generated Response (Expected):

### Step 1: Definition of Fibonacci Sequence

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1.

The sequence begins: 0, 1, 1, 2, 3, 5, 8, 13, 21, and so on.

### Step 2: JavaScript Function

```
function fibonacci(n) {
 if (n <= 0) return 0;
 if (n === 1) return 1;

 let a = 0, b = 1;
 for (let i = 2; i <= n; i++) {
 let next = a + b;
 a = b;
 b = next;
 }
 return b;
}
```

### Step 3: Explanation of Logic

- **Base Cases:** The function first handles the base cases: if `n` is 0, it returns 0; if `n` is 1, it returns 1. These are the starting points of the sequence.
- **Iterative Approach:** For `n` greater than 1, an iterative approach is used. Two variables, `a` and `b`, are initialised to 0 and 1, respectively, representing the previous two Fibonacci numbers.
- **Looping:** The `for` loop starts from `i = 2` up to `n`. In each iteration:
  - `next` calculates the sum of `a` and `b` (the following Fibonacci number).
  - `a` is updated to the value of `b`.
  - `b` is updated to the value of `next`.
- **Return Value:** After the loop completes, `b` will hold the nth Fibonacci number.

- **Analysis:** The AI provides a clear, step-by-step explanation along with the code, making it easier to understand the solution and verify its correctness.



# Role-Playing and Persona: Guiding the AI's Perspective

---

Assigning a specific role or persona to the AI can influence its tone, style, and the type of code it generates. This is useful when you need code that adheres to specific professional standards or has a particular focus (e.g., security, performance).

## Example 3: Security-Focused Code

**Goal:** Generate a function to sanitise user input, with a focus on security.

**Prompt:**

```
Act as a cybersecurity expert. Write a JavaScript function `sanitizeInput`
```

**AI-Generated Response (Expected):**

```
function sanitizeInput(input) {
 // Create a new DOM parser to safely parse the input string
 const doc = new DOMParser().parseFromString(input, 'text/html');

 // Create a temporary div to hold the parsed content
 const div = document.createElement('div');
 div.appendChild(doc.body);

 // Remove script tags and attributes that could lead to XSS
 // This is a simplified example; a robust solution would use a library
 const scripts = div.getElementsByTagName('script');
 for (let i = scripts.length - 1; i >= 0; i--) {
 scripts[i].parentNode.removeChild(scripts[i]);
 }

 // Remove event handlers (e.g., onclick, onerror) from all elements
 const allElements = div.querySelectorAll('*');
 allElements.forEach(element => {
 Array.from(element.attributes).forEach(attr => {
 if (attr.name.startsWith('on')) {
 element.removeAttribute(attr.name);
 }
 });
 });

 return div.innerHTML;
}
```

```
/*
Security Considerations:

1. Client-Side vs. Server-Side: Client-side sanitisation (like this example) is a good
line of defence but is NOT sufficient on its own.
Server-side sanitisation is crucial because malicious actors can bypass client-side
defences.

2. Complexity of XSS: XSS attacks can be very sophisticated. Manually implementing
sanitisation can be prone to errors and incomplete.
For production applications, it is highly recommended to use well-vetted
sanitisation libraries (e.g., DOMPurify for client-side, or server-side libraries like
appropriate for your backend language).

3. Whitelisting vs. Blacklisting: This simplified example uses a form of blacklisting
(removing known harmful elements/attributes). A more secure approach is whitelisting,
where you only allow known safe elements and attributes.

4. Contextual Escaping: Different contexts (HTML, attributes, JavaScript) require
different escaping mechanisms. A general-purpose sanitiser might not handle all
edge cases.
*/
```

- **Analysis:** The AI not only provides a function but also explains the security implications and best practices, demonstrating its understanding of the

# Chapter 12: Best Practices in AI-Assisted Coding – Collaboration and Ethics

## The Human-AI Partnership: Beyond Code Generation

---

As you become more proficient in prompting AI for code, it's crucial to understand that this is not just about generating lines of code. It's about a new form of collaboration between humans and machines. Just as Forth emphasises the interaction between the programmer and the system, AI-assisted coding thrives on a symbiotic relationship where the AI augments your capabilities, and you guide its intelligence. This chapter will explore best practices for human-AI partnerships, focusing on collaboration workflows, ethical considerations, and the responsible use of AI.

## Collaborative Workflows with AI

---

Integrating AI into your coding workflow isn't about replacing you; it's about making you more efficient and effective. Here are some best practices for collaborating with your AI assistant:

1. **Start with Clear Intent:** Before prompting, have a clear idea of what you want to achieve. A well-defined problem statement is the foundation for a good prompt.
2. **Iterate and Refine Prompts:** Treat prompting as a conversation. Don't expect perfect code on the first try. Refine your prompts based on the AI's responses, providing more context, constraints, or examples as needed.
3. **Review and Understand Generated Code:** Never blindly trust AI-generated code. Always review it for correctness, efficiency, security, and adherence to your project's standards. Understand *why* the AI generated a particular solution.
4. **Test Thoroughly:** Just as with any other code, AI-generated code requires rigorous testing. Write unit tests, integration tests, and perform manual testing to ensure it behaves as expected.
5. **Integrate Incrementally:** For larger features, integrate AI-generated code in small, manageable chunks. This makes it easier to debug and verify the code.

6. **Use AI for Repetitive Tasks:** Leverage AI for tasks such as boilerplate code, simple functions, regular expressions, or data format conversions. This frees you up for more complex problem-solving.
7. **Learn from AI:** Pay attention to the patterns, libraries, or approaches the AI suggests. It can expose you to new techniques and best practices that you might not have been aware of.
8. **Maintain Version Control:** Utilise Git or another version control system. This allows you to track changes, revert to previous versions, and collaborate effectively, even when AI is involved.

## Ethical Considerations in AI-Assisted Coding

---

As AI becomes more integrated into software development, new ethical questions arise. Responsible use of AI is paramount.

1. **Bias in Training Data:** AI models are trained on vast datasets of existing code. If this data contains biases (e.g., favouring certain coding styles, languages, or even reflecting historical biases in human code), the AI might perpetuate or amplify them. Be aware of this potential and actively work to mitigate it.
2. **Intellectual Property and Licensing:** The code generated by AI is derived from its training data. The legal implications regarding intellectual property, copyright, and licensing of AI-generated code are still evolving. Be mindful of the licenses of the code snippets you use, especially if they are from open-source projects.

### Prompt:

```
Explain the concept of open-source software licenses (e.g., MIT, GPL) and the legal implications for using AI-generated code that might be derived from such licenses.
```

### AI-Generated Explanation (Expected):

Open-source software licenses dictate how software can be used, modified, and distributed. Key licenses include:

\* **MIT License:** A permissive license, allowing almost unrestricted use and distribution, provided the original copyright and license notice are preserved. Code derived from MIT-licensed projects can often be used in proprietary software.

\* **GPL (General Public License):** A copyleft license that requires any code that is modified or built upon to also be distributed under the GPL. If you incorporate GPL-licensed code into your project (including code generated by AI that is based on GPL code), your project may need to be open-sourced under the same license.

\* **Apache License 2.0:** Similar to MIT but with explicit patent rights for preserving notices and modifications.

\* **BSD Licenses:** A family of permissive licenses with minimal restrictions. They are similar to MIT.

When using AI-generated code, it's important to consider that the AI model is trained on code under various licenses. If the generated code closely resembles or is derived from open-source projects, you may be obligated to comply with those license terms, particularly for copyleft licenses such as the GPL.

3. **Attribution and Credit:** Some licenses require attribution to the original authors. Even if the code is generated by AI, you may need to provide credit if the output is substantially similar to existing open-source code.
4. **Security and Privacy:** AI-generated code can inadvertently introduce vulnerabilities or expose sensitive information. Always review code for security risks and avoid sharing confidential data in prompts.
5. **Transparency:** Be transparent with your team or stakeholders when using AI-generated code. Document which parts of your codebase were AI-assisted and any relevant license or attribution requirements.

## Responsible Use: Your Role as a Developer

---

AI is a powerful tool, but with great power comes great responsibility. Here are some guiding principles for responsible AI-assisted coding:

- **Stay Informed:** Keep up to date with the evolving legal and ethical landscape around AI and software development.
- **Educate Others:** Share your knowledge of best practices and ethical considerations with your peers.

- **Promote Inclusivity:** Utilise AI to enhance the accessibility and inclusivity of your code, and be aware of potential biases that may be inherent in AI-generated solutions.
- **Own Your Code:** Ultimately, you are responsible for the code you ship. Use AI as a collaborator, not a replacement for your own judgment and expertise.

## Summary

---

The future of coding is collaborative. By combining your creativity, critical thinking, and ethical awareness with the speed and breadth of AI, you can build better software and help shape a responsible, inclusive, and innovative future for all developers.

# Chapter 13: Building a Real-World Application - The To-Do List

## Bringing It All Together: Your First Complete Project

---

In *Starting Forth*, you learned how to combine individual words into more complex definitions and ultimately build applications. Now, it's time to apply all the JavaScript knowledge and AI prompting techniques you've acquired to construct a complete, functional **To-Do List application**. This project will integrate concepts from previous chapters, including DOM manipulation, event handling, data structures, and perhaps even local storage for persistence.

This chapter guides you through building a simple web-based To-Do List application step-by-step, utilising AI to generate the necessary HTML, CSS, and JavaScript code. We will focus on the core functionalities:

- Adding new tasks
- Marking tasks as complete
- Deleting tasks
- (Optional) Persisting tasks using Local Storage

## Project Setup: HTML and CSS

---

Every web application starts with its structure (HTML) and presentation (CSS). We'll prompt the AI to create a basic layout for our To-Do List.

### Example 1: Basic HTML Structure

**Goal:** Create the fundamental HTML for a To-Do List application, including an input field for new tasks, an add button, and an unordered list to display tasks.

**Prompt:**

Generate the basic HTML structure for a simple To-Do List web application. It should include:

- A main heading (h1) with the title "My To-Do List"
- An input field for new tasks (id="taskInput")
- A button to add tasks (id="addTaskBtn") with the text "Add Task"
- An empty unordered list (ul) with the id="taskList" where tasks will be added

### AI-Generated Code (Expected):

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <title>My To-Do List</title>
 <link rel="stylesheet" href="style.css"> <!-- Link to our CSS file -->
</head>
<body>
 <h1>My To-Do List</h1>
 <div>
 <input type="text" id="taskInput" placeholder="Add a new task..." />
 <button id="addTaskBtn">Add Task</button>
 </div>
 <ul id="taskList">

 <script src="script.js"></script> <!-- Link to our JavaScript file -->
</body>
</html>
```

- **Analysis:** The AI provides a clean, semantic HTML structure, including placeholders for our CSS and JavaScript files.

## Example 2: Basic CSS Styling

**Goal:** Add some basic CSS to make the To-Do List visually appealing.

### Prompt:

Provide basic CSS for the To-Do List HTML structure. Set the body to a light blue color, centre the main content, style the input field and button, and add padding to the list items. Also, style completed tasks with a line-through text decoration.

### AI-Generated Code (Expected):

```
body {
 font-family: Arial, sans-serif;
```



```
 background-color: #f4f4f4;
 display: flex;
 justify-content: center;
 align-items: flex-start; /* Align to top */
 min-height: 100vh;
 margin: 20px;
}

h1 {
 text-align: center;
 color: #333;
}

div {
 margin-bottom: 20px;
 display: flex;
 gap: 10px;
}

input[type="text"] {
 padding: 10px;
 border: 1px solid #ddd;
 border-radius: 4px;
 flex-grow: 1;
}

button {
 padding: 10px 15px;
 background-color: #007bff;
 color: white;
 border: none;
 border-radius: 4px;
 cursor: pointer;
}

button:hover {
 background-color: #0056b3;
}

ul {
 list-style: none;
 padding: 0;
 width: 100%;
 max-width: 500px; /* Limit width for better readability */
}

li {
 background-color: white;
```

```
padding: 10px 15px;
margin-bottom: 8px;
border-radius: 4px;
display: flex;
justify-content: space-between;
align-items: center;
box-shadow: 0 2px 4px rgba(0,0,0,0.1);
}

li.completed {
 text-decoration: line-through;
 color: #888;
 background-color: #e9e9e9;
}

li button {
 background-color: #dc3545; /* Red for delete button */
 padding: 5px 10px;
 font-size: 0.8em;
}

li button:hover {
 background-color: #c82333;
}
```

- **Analysis:** The AI provides well-structured CSS that makes the application look clean and functional, including a class for completed tasks.

## Core Functionality: JavaScript

---

Now for the main logic. We'll prompt the AI to handle adding, completing, and deleting tasks.

### Example 3: Adding New Tasks

**Goal:** Implement the JavaScript to add a new task when the button is clicked.

**Prompt:**

Write JavaScript code for the To-Do List application.

1. Get references to the ``taskInput``, ``addTaskBtn``, and ``taskList`` elements.
2. Add an event listener to ``addTaskBtn`` for a `'click'` event.
3. Inside the event listener:
  - a. Get the value from ``taskInput``.
  - b. If the input is not empty, create a new ``li`` element.
  - c. Set the ``textContent`` of the ``li`` to the task input value.
  - d. Append the new ``li`` to the ``taskList``.
  - e. Clear the ``taskInput`` field.

### AI-Generated Code (Expected):

```
// script.js
const taskInput = document.getElementById("taskInput");
const addTaskBtn = document.getElementById("addTaskBtn");
const taskList = document.getElementById("taskList");

addTaskBtn.addEventListener("click", function() {
 const taskText = taskInput.value.trim(); // .trim() removes leading/trailing whitespace

 if (taskText !== "") {
 const listItem = document.createElement("li");
 listItem.textContent = taskText;
 taskList.appendChild(listItem);
 taskInput.value = ""; // Clear input field
 }
});
```

- **Analysis:** The AI correctly implements the logic for adding tasks, including trimming whitespace and clearing the input field.

## Example 4: Marking Tasks as Complete and Deleting Tasks

**Goal:** Add functionality to mark tasks as complete and delete them.

**Prompt:**

Extend the JavaScript for the To-Do List. For each new task (and existing ones were to load them):

1. Add a 'click' event listener to each ``li`` element. When clicked, toggle the `completed` CSS class on that ``li``.
2. Add a 'Delete' button inside each ``li`` element.
3. Add a 'click' event listener to the 'Delete' button. When clicked, remove the ``li`` element from the DOM.

Modify the task creation logic to include these features.

### **AI-Generated Code (Expected):**

```
// script.js (updated)
const taskInput = document.getElementById("taskInput");
const addTaskBtn = document.getElementById("addTaskBtn");
const taskList = document.getElementById("taskList");

function createTaskElement(taskText) {
 const listItem = document.createElement("li");
 listItem.textContent = taskText;

 // Add click listener to toggle 'completed' class
 listItem.addEventListener("click", function() {
 listItem.classList.toggle("completed");
 });

 // Create Delete button
 const deleteBtn = document.createElement("button");
 deleteBtn.textContent = "Delete";
 deleteBtn.addEventListener("click", function(event) {
 event.stopPropagation(); // Prevent listItem click event from firing
 listItem.remove(); // Remove the li element
 });

 listItem.appendChild(deleteBtn);
 return listItem;
}

addTaskBtn.addEventListener("click", function() {
 const taskText = taskInput.value.trim();

 if (taskText !== "") {
 const newListItem = createTaskElement(taskText);
 taskList.appendChild(newListItem);
 taskInput.value = "";
 }
});
```

- **Analysis:** The AI refactors the task creation into a separate function, adds the `completed` class toggling, and includes a functional delete button. The `event.stopPropagation()` is a crucial detail to prevent unintended behaviour.

## Example 5 (Optional): Persisting Tasks with Local Storage

**Goal:** Make the To-Do List tasks persist even after the browser is closed.

**Prompt:**

Modify the To-Do List JavaScript to save and load tasks from the browser

1. When a task is added, completed, or deleted, save the current list of tasks to Local Storage.
2. When the page loads, load tasks from Local Storage and display them.

Each task should be stored as an object with `text` and `completed` properties.

### AI-Generated Code (Expected):

```
// script.js (further updated)
const taskInput = document.getElementById("taskInput");
const addTaskBtn = document.getElementById("addTaskBtn");
const taskList = document.getElementById("taskList");

// --- Local Storage Functions ---
function saveTasks() {
 const tasks = [];
 taskList.querySelectorAll("li").forEach(listItem => {
 tasks.push({
 text: listItem.textContent.replace("Delete", "").trim(),
 // Remove 'Delete' text
 completed: listItem.classList.contains("completed")
 });
 });
 localStorage.setItem("todoTasks", JSON.stringify(tasks));
}

function loadTasks() {
 const storedTasks = localStorage.getItem("todoTasks");
 if (storedTasks) {
 const tasks = JSON.parse(storedTasks);
 tasks.forEach(task => {
 const newListItem = createTaskElement(task.text);
 if (task.completed) {
 newListItem.classList.add("completed");
 }
 taskList.appendChild(newListItem);
 });
 }
}

// --- Task Creation and Event Handling ---
function createTaskElement(taskData) {
 const listItem = document.createElement("li");
 // Handle if taskData is just text or an object
 const taskText = typeof taskData === 'object' ? taskData.text : taskData;
```

```

 listItem.textContent = taskText;

 // Add click listener to toggle 'completed' class
 listItem.addEventListener("click", function() {
 listItem.classList.toggle("completed");
 saveTasks(); // Save after status change
 });

 // Create Delete button
 const deleteBtn = document.createElement("button");
 deleteBtn.textContent = "Delete";
 deleteBtn.addEventListener("click", function(event) {
 event.stopPropagation();
 listItem.remove();
 saveTasks(); // Save after deletion
 });

 listItem.appendChild(deleteBtn);
 return listItem;
}

addTaskBtn.addEventListener("click", function() {
 const taskText = taskInput.value.trim();

 if (taskText !== "") {
 const newListItem = createTaskElement(taskText);
 taskList.appendChild(newListItem);
 taskInput.value = "";
 saveTasks(); // Save after adding new task
 }
});

// Load tasks when the page first loads
document.addEventListener("DOMContentLoaded", loadTasks);

```

- **Analysis:** This advanced example demonstrates the use of `localStorage` to persist data, `JSON.stringify()` and `JSON.parse()` for data serialisation, and integrating saving and loading into the existing event handlers. It also shows how to handle the `EventListener`.

# Chapter 14: Beyond the Browser - JavaScript on the Server and Desktop

## JavaScript Everywhere: Node.js and Desktop Applications

---

JavaScript, once confined to the browser, has broken free and now powers a vast ecosystem of applications beyond the web. This chapter will introduce you to **Node.js**, which enables JavaScript to run on servers, and touch upon frameworks that allow JavaScript to be used in **desktop applications**. You'll learn how to prompt AI to generate code for these environments, expanding your reach as a JavaScript developer.

### Node.js: JavaScript on the Server

---

**Node.js** is a JavaScript runtime environment built on top of Chrome's V8 JavaScript engine. It allows you to run JavaScript code outside of a web browser, typically for server-side applications, command-line tools, and backend services. This means you can use a single language (JavaScript) for both your frontend (browser-based) and backend (server-based) development, a concept known as "full-stack JavaScript."

### Setting up Node.js on an External Server

To deploy your Node.js application, you'll typically need a server. Here's a simplified guide:

1. **Choose a Hosting Provider:** Select a hosting provider that supports Node.js. Popular options include Heroku, Amazon Web Services (AWS), Google Cloud, DigitalOcean, and Vultr.
2. **Set up a Server:**
  - **For VPS (Virtual Private Server):** You'll get full control over the server. Install Node.js using a package manager like `apt` (on Debian/Ubuntu) or `yum` (on CentOS/RHEL). Example:

```
sudo apt update
sudo apt install nodejs npm
```



- **For Platform-as-a-Service (PaaS):** Services like Heroku simplify deployment. You typically push your code to their platform, and they handle the server setup.
3. **Transfer Your Code:** Use tools like `git` or `scp` to transfer your Node.js application files to the server.
  4. **Install Dependencies:** Navigate to your application directory on the server and run:

```
npm install
```

This command installs all the necessary packages listed in your `package.json` file.

5. **Start Your Application:** Use a process manager like `pm2` or `forever` to keep your Node.js application running even if the server is restarted. Example:

```
npm install -g pm2
pm2 start your_app.js
```

## Working with Shared Hosting (No Bash Access)

If you're limited to a shared hosting environment without direct Bash access, using Node.js can be tricky, but not impossible. Here are a few strategies:

1. **Check for Node.js Support:** Some shared hosting providers offer limited Node.js support through their control panel (e.g., cPanel). Look for options to run Node.js applications.
2. **Use a Pre-built Application:** If your hosting provider allows it, deploy a pre-built Node.js application (such as a simple Express server) via FTP or their file manager. Then, configure it through the available control panel settings.
3. **Consider Serverless Functions:** If shared hosting is a must, explore serverless functions (like AWS Lambda or Netlify Functions). These allow you to run JavaScript code on the server without needing to manage the server itself. You can trigger these functions via HTTP requests from your frontend.

## Desktop Applications with JavaScript

JavaScript isn't just for web servers; it can also be used to create desktop applications. Frameworks like **Electron** and **NW.js** allow you to build cross-platform desktop apps using web technologies (HTML, CSS, JavaScript).

1. **Electron:** Utilises Chromium and Node.js to develop desktop applications. Popular for apps like VS Code, Slack, and Discord.
2. **NW.js (Node-Webkit):** Another framework for building desktop apps with web

technologies.

## Bundling for Stand-alone Apps

To distribute your desktop application, you'll need to bundle it into a stand-alone executable. Both Electron and NW.js provide tools for this:

- **Electron:** Use `electron-packager` or `electron-builder` to package your app for different platforms (Windows, macOS, Linux).
- **NW.js:** Provides a command-line tool to bundle your application into a single executable.

These tools package your application code, the Node.js runtime, and the necessary Chromium files into a single, distributable package.

## Prompting AI for Server and Desktop Code

---

Now, let's see how you can use AI to generate code for Node.js and desktop applications.

### Example 1: Creating a Simple Node.js Server

**Goal:** Generate a basic HTTP server using Node.js and Express.

**Prompt:**

```
Generate a Node.js application using Express that listens on port 3000 and responds with "Hello, World!" to all requests. Include comments to explain each step.
```

**AI-Generated Code (Expected):**

```
const express = require('express'); // Import the Express library
const app = express(); // Create an Express application
const port = 3000; // Define the port number

// Define a route that responds with "Hello, World!" for all requests.
app.get('/', (req, res) => {
 res.send('Hello, World!');
});

// Start the server and listen on the specified port.
app.listen(port, () => {
 console.log(`Server listening at http://localhost:${port}`);
});
```

## Example 2: Building a Simple Desktop App with Electron

**Goal:** Generate a basic Electron application that displays "Hello, World!" in a window.

**Prompt:**

Generate a basic Electron application that creates a window and displays "Hello, World!" in the window. Include comments to explain each step.

**AI-Generated Code (Expected):**

```
const { app, BrowserWindow } = require('electron');

function createWindow () {
 // Create a new browser window
 let win = new BrowserWindow({
 width: 800,
 height: 600,
 webPreferences: {
 nodeIntegration: true
 }
 });

 // Load the HTML file
 win.loadFile('index.html');
}

// When the app is ready, create the window
app.on('ready', createWindow);

// HTML file (index.html):
// <!DOCTYPE html>
// <html>
// <head>
// <title>Hello, World!</title>
// </head>
// <body>
// <h1>Hello, World!</h1>
// </body>
// </html>
```

### Summary

JavaScript's versatility extends far beyond the browser. With Node.js, you can build powerful server-side applications, and with frameworks like Electron, you can create cross-platform desktop apps. By leveraging AI to generate code for these environments, you can significantly expand your capabilities as a JavaScript developer.

# Chapter 15: The Journey Continues - Next Steps in AI-Assisted Development

## Your Path Forward: Continuous Learning and Exploration

---

Congratulations! You have completed your journey through "Prompting for Code." This is a beginning, not an end. This book has equipped you with the foundational knowledge and practical skills to effectively leverage AI in your JavaScript development. You've learned to communicate with AI, generate code, refine it, build applications, and even explore JavaScript beyond the browser. However, the world of AI and software development is constantly evolving, and your journey as a developer is an ongoing one.

This final chapter will guide you on how to continue your learning, stay updated with the latest advancements, and further integrate AI into your development workflow.

## Staying Updated in a Rapidly Evolving Field

---

The landscape of AI tools and programming languages changes rapidly. To remain effective and competitive, continuous learning is essential.

1. **Follow AI/ML News and Blogs:** Subscribe to reputable newsletters, blogs, and news sources that cover AI, machine learning, and software development. Key sources include:
  - **Google AI Blog:** For research and applications from Google.
  - **OpenAI Blog:** For updates on GPT models and other AI research.
  - **Towards Data Science (Medium):** A popular platform for articles on AI, ML, and data science.
  - **Hacker News:** For general tech news and discussions.
2. **Experiment with New AI Tools:** Don't be afraid to try out new AI code generation tools as they emerge. Each tool may have its strengths and weaknesses, and experimentation will help you determine what works best for your specific needs.
3. **Participate in Online Communities:** Join developer communities on platforms such as

Stack Overflow, GitHub, Reddit (e.g., r/learnjavascript, r/MachineLearning, r/promptengineering), and Discord. These communities are excellent for asking questions, sharing knowledge, and staying informed.

4. **Read Documentation and Research Papers:** For a deeper understanding, delve into the official documentation of AI models and tools, and if you're inclined, explore relevant research papers.

## Deepening Your JavaScript Knowledge

---

While this book focused on Vanilla JavaScript, the ecosystem is vast. Consider exploring:

1. **Frameworks and Libraries:**
  - **React, Angular, Vue.js:** Popular frontend frameworks for building complex user interfaces.
  - **Next.js, Nuxt.js:** Frameworks built on React/Vue for server-side rendering and static site generation.
  - **Express.js:** A minimalist web framework for Node.js, ideal for building APIs.
2. **TypeScript:** A superset of JavaScript that adds static typing, which can significantly improve code quality and maintainability, especially in larger projects.
3. **Testing Frameworks:** Learn how to write automated tests for your JavaScript code using frameworks like Jest, Mocha, or Cypress.
4. **Build Tools:** Familiarise yourself with tools such as Webpack, Rollup, or Vite for bundling and optimising your JavaScript applications.

## Advanced AI-Assisted Development Strategies

---

As you gain experience, you can employ more sophisticated strategies for AI collaboration:

1. **Fine-tuning Models (for advanced users):** If you have specific coding patterns or domain-specific knowledge, you might explore fine-tuning smaller AI models on your codebase to generate even more tailored and accurate code.
2. **Automated Code Review with AI:** Utilise AI tools to automatically review your code for potential bugs, security vulnerabilities, and style inconsistencies.
3. **AI for Documentation and Explanations:** Leverage AI to generate documentation for your code or to explain complex algorithms and concepts.
4. **AI for Code Migration and Refactoring:** Utilise AI to aid in migrating legacy

codebases to newer versions of JavaScript or to refactor existing code for improved performance and readability.

## The Ethical Developer

---

Remember the ethical considerations discussed in Chapter 12. As AI tools become increasingly powerful, your role as a developer also grows. Always strive to:

- **Be Transparent:** Disclose when AI has been used to generate code, especially in collaborative environments.
- **Prioritise Security:** Never deploy AI-generated code without a thorough security review and testing.
- **Promote Fairness and Inclusivity:** Be mindful of potential biases in AI models and work towards creating equitable and accessible software.

## Conclusion: The Future is Collaborative

---

The journey of learning to code with AI is an exciting one. You are at the forefront of a technological shift that is redefining how software is built. Embrace the power of AI as your intelligent assistant, but never forget that your creativity, critical thinking, and problem-solving skills remain indispensable. Continue to learn, experiment, and build, and you will undoubtedly shape the future of software development.

Happy Prompting, and Happy Coding!



### References:

- [Node.js Official Website](#)
- [React Official Website](#)
- [Angular Official Website](#)
- [Vue.js Official Website](#)
- [TypeScript Official Website](#)
- [Jest Testing Framework](#)

