NAVAIR SOFTWARE ENABLEMENT TEAM

The Warfighter-Coder Handbook: Web App

From Problem to Deployed Tool

(Version 1)

Authored by: Sean Lavelle

Edited by: Zach Fisher

The mission of this handbook is to empower you to solve real-world problems by building simple, secure software with AI, even if you've never written a line of code. This guide is for the motivated problem-solver who sees a better way.

The character of conflict is changing. The speed at which we must adapt is accelerating, and the gap between identifying a tactical problem and deploying a technical solution is often too wide. The warfighter on the ground, at sea, or in the air has the most immediate and intimate understanding of the challenges they face. This guide is built on the belief that those closest to the problem are best equipped to solve it. This is the strategic imperative for creating a cadre of warfighter-coders: to equip our most valuable asset—our people—with the skills to build their own solutions at the speed of relevance.

This handbook will not teach you to become a professional developer. Instead, it will teach you a more critical skill: how to become a master problem-solver who uses Artificial Intelligence as a tool. You will learn to manage an AI "pair programmer" to write the code for you, allowing you to focus on what truly matters.

This requires a shift in thinking to a **"problem-first, tool-second" mindset**. The technology is a powerful enabler, but it is not the mission. Your primary skill will be your ability to observe a workflow, identify a point of friction, and define the problem with such clarity that even a machine can understand the objective. You will learn to build simple, secure, and powerful offline tools that work in the most constrained environments, directly addressing the needs you see every day.

This guide is structured to take you from core concepts to a finished product. **Part 1: The Building Blocks** will cover the foundational "computer science stuff"—what an LLM is, the three languages of the web, and how to structure your files and data. **Part 2: The Core Workflow** gets hands-on, guiding you through the process of defining a problem, writing effective prompts, and iterating on your first application. Later sections will cover more advanced topics like adding memory to your apps and preparing them for deployment.

Before you begin, there is one paramount rule. The tools we use are powerful, but they require discipline. For this reason, we ask that you **please read Chapter 6 about security before building anything work-related with an LLM.** It is the most important chapter in this handbook. Understanding the operational security (OPSEC) principles of working with AI is not optional; it is the foundation upon which everything else is built.

Welcome to the mission.

Table of Contents

Foreword: The Mission....1

Part 1: The Building Blocks

Chapter 1: Your New 'Pair Programmer': What is an LLM? 3

Chapter 2: Your New Capability: Building Software Today 4

Chapter 3: The Three Languages of the Web 5

Chapter 4: Your Project's Blueprint: File Structure 8

Chapter 5: Understanding Data Formats: JSON and CSV 11

Part 2: The Core Workflow

Chapter 6: The Golden Rule: **OPSEC First 13

Chapter 7: Finding the Real Problem (The TAG Framework & JTBD) 15

Chapter 8: How to Talk to Your AI (The TCF Prompt Pattern) 18

Part 3: The Build Process

Chapter 9: Your First Build: The Checklist App 22

Chapter 10: The Build Loop: Test, Triage, and Refine 24

Chapter 11: Making Your App Remember: `localStorage` 26

Chapter 12: The Data Lifeline: Import & Export 29

Part 4: Hardening & Deployment

Chapter 13: Proving Your App is Secure (The Code Audit) 31

Chapter 14: Preparing for Approval: Your Documentation Packet 34

Chapter 15: From Folder to Single File for Distribution 37

Chapter 16: Versioning & Distribution 40

Part 5: Advanced Capabilities

Chapter 17: Multi-User Collaboration with ShareDrive-NoSQL 42

Chapter 18: Adding a Map: Offline GeoJSON with Leaflet 45

Part 6: Reference & Next Steps

Chapter 19: The Pattern Library & Troubleshooting 49

Chapter 20: Your Force-Multiplier: The Warfighter Coder Tool (WCT) 53

Chapter 21: Your 30-Day Mission 56

Appendices

Appendix A: Master Prompt Library 58

Appendix B: All Checklists 62

Appendix C: Example Approval Packet 64

Part 1: The Building Blocks

(Focus: Establishes the foundational concepts you need before writing a single prompt. This part gets you ready to build.)

Chapter 1: Your New 'Pair Programmer': What is an LLM?

Concept: This is human-machine teaming

A Large Language Model (LLM) is an artificial intelligence trained on vast amounts of internet text and code. Built on a concept called a neural network, its primary function is to predict the next word in a sequence. By learning the patterns of language and code, it can generate everything from sentences to entire applications. Think of it not as a thinking entity, but as the world's most advanced autocomplete, calculating the most probable output based on your prompt.

For our purposes, we'll treat the LLM as a super-fast junior coder. Its training allows it to generate functional code in any language within seconds, a speed that is its primary superpower.

However, this speed comes with critical limitations. The LLM has **zero context** about your mission or security constraints. It only follows your instructions and can "hallucinate"—inventing plausible but non-functional code. As a confident but flawed assistant, it will execute illogical orders without question. Your job is to verify everything.

This makes your role the **manager, strategist, and tester**, not the typist. Your mission is to provide operational context, break down problems into clear prompts, and rigorously test the AI-generated code. You provide specific feedback in an iterative loop until the tool is correct and reliable.

These limitations are a moving target. AI is advancing at an unprecedented rate, with new models showing dramatic improvements in accuracy and contextual understanding. The 'junior coder' of today is effectively getting promoted every few months. Your role will evolve from catching basic errors to directing a more capable partner on increasingly complex problems.

This 'pair programmer' model is a partnership where you are in command. The LLM provides speed, but you provide the direction, judgment, and validation. The following chapters will teach you the methods to manage this partnership, guiding you from problem to deployed tool and giving your team a decisive edge.

How Large Language Models Work by IBM Technology [http://www.youtube.com/watch?v=5sLYAQS9sWQ]

Large Language Models explained briefly by 3Blue1Brown [http://www.youtube.com/watch?v=LPZh9BOjkQs]

What are Large Language Models (LLMs)? by Google for Developers [http://www.youtube.com/watch?v=iR2O2GPbB0E]

Chapter 2: Your New Capability: Building Software Today

> **The Big Idea: You don't need to be a professional coder. We build simple, offline tools that solve real problems, period.**

This guide is built on a single, powerful premise: any motivated warfighter can build software to solve immediate, real-world problems. You don't need a computer science degree or years of experience. By leveraging the right mindset and modern AI tools, you can create and deploy useful applications today. We will focus exclusively on building simple, powerful, and secure "static web apps" that run entirely on a local machine without needing a server or an internet connection. This approach is a strategic choice for our operating environment.

Why Offline First? The Tactical Advantage

Choosing to build offline applications gives us three immediate advantages that are perfectly suited for the DoD environment:

- **Zero Installation:** An offline app is just a file (or a folder of files). There is nothing to install, and no admin rights are required. The end-user simply double-clicks an `html` file, and it opens instantly in their web browser. This eliminates one of the biggest hurdles to deploying software in a locked-down environment. Double-click and go.
- **Secure by Design:** Our applications have no server to hack and make no network calls to intercept. By design, they cannot "phone home" or expose data over a network. This dramatically reduces the cybersecurity attack surface, making the approval and deployment process significantly simpler and faster.
- **Infinitely Portable:** A finished tool can be packaged into a single file or a simple ZIP folder. It can be distributed easily and securely through command-approved channels like SharePoint, email, or trusted media transfer. This ensures that the tools you build can get into the hands of those who need them, regardless of network conditions.

While this offline-first approach provides a powerful and secure foundation, it is just the beginning. The skills you learn building these standalone tools are the building blocks for more advanced capabilities. In later sections, we will explore how to evolve these personal applications into collaborative tools that can read from and write to a shared data source, such as a file on a SharePoint site, enabling an entire team to work together.

Make a static site work offline (by hand) - Progressive Web App Training by Chrome for Developers [http://www.youtube.com/watch?v=dXuvT4oolIQ]

Progressive Web Apps in 100 Seconds // Build a PWA from Scratch by Fireship [http://www.youtube.com/watch?v=sFsRylCQblw]

Chapter 3: The Three Languages of the Web

> **Concept: It helps to understand what the code does–like knowing your equipment.**

Your primary job is not to become a professional programmer, but to effectively manage the AI that writes the code. This requires a basic ability to read and understand the materials your AI gives you, just as a commander needs to know the function of their equipment to lead effectively. Web applications, from simple tools to complex websites, are built using three core languages that work together in concert.

The simplest way to think of this is like building a house. **HTML** is the structure - the foundation, walls, and roof. **CSS** is the style and decoration - the paint, flooring, and furniture that make the house look good. Finally, **JavaScript** is the functionality - the electrical wiring, plumbing, and appliances that make the house *do* things.

Your `index.html` file acts as the master blueprint. It contains the HTML structure, but it also holds the CSS rules (typically inside a `<style>` tag) and the JavaScript logic (inside a `<script>` tag). The browser reads this single file, understands how all three parts connect, and assembles the final, interactive tool you see on the screen. The JavaScript can find an HTML element, and when a user clicks it, instantly change its text or even modify its CSS styling. This interplay is what turns a static page into a functional application.

1. HTML: The Structure (The Blueprint)

HTML (HyperText Markup Language) is the skeleton of your application. It provides the raw content and defines the layout using a system of **tags** enclosed in angle brackets. Most tags come in pairs: an opening tag like `<p>` and a closing tag like `</p>`. The content between them is what the browser displays.

Some tags are for structure, like `<div>`, which is a generic container used to group other elements. Others are for content, like `<h1>` for a main heading or `<p>` for a paragraph. Interactive elements also have their own tags, like `<button>`.

To make these elements unique so our other languages can find them, we use **attributes**. The two most important attributes are `id` and `class`.

- An `id` is a unique name for **one specific element**, like a serial number.
- A `class` is a label you can apply to **many elements** to group them together.

Example HTML Block:

```
<!-- A simple container for our app with a "class" -->
<div class="card">
    <h1>Mission Prep Checklist</h1>
    <!-- A paragraph with a unique "id" -->
    <p id="status-text">Status: Incomplete</p>
    <button id="complete-mission-btn">Mark Complete</button>
</div>
```

What to look for: Your main job is to scan the HTML tags to get a quick mental map of the page's structure and find the `id` or `class` attributes that JavaScript and CSS will use as hooks.

To learn more: Watch this video introduction: HTML Crash Course (https://www.youtube.com/watch%3Fv%3DUB1O30fR-EE)

2. CSS: The Style (The Interior Decorator)

CSS (Cascading Style Sheets) makes your HTML look good. It works by "selecting" HTML elements using their tag name, class, or id, and then applying styling rules to them. The basic syntax is `selector { property: value; }`. To target an element by its class, you use a period (`.`). To target an element by its unique id, you use a hash (`#`). This is how CSS connects to the HTML structure you defined.

Example CSS Block (styling the HTML above):

```
/* Style any element with the "card" class */
.card {
    background-color: #f0f0f0; /* A light gray */
    border: 1px solid #ccc;
    padding: 20px; /* Space inside the border */
}

/* Style the one element with the "complete-mission-btn" id */
#complete-mission-btn {
    background-color: #007bff; /* A nice blue */
    color: white;
}
```

What to look for: When reading CSS, first find the selector (`.card`, `#complete-mission-btn`) to know *what* is being styled. Then, look at the properties inside the curly braces to understand *how* it's being styled.

To learn more: For a video tutorial, see this CSS Crash Course (https://www.youtube.com/watch%3Fv%3DyfoY53QXEnI)

3. JavaScript: The Action (The Electrician & Plumber)

JavaScript (JS) is what brings your application to life. It's a programming language that can manipulate your HTML and CSS in response to user actions like clicks or key presses. When you click a button and something happens, that's JavaScript at work.

It operates on the **Document Object Model (DOM)**, which is just the browser's live, interactive model of your HTML structure. JavaScript can ask the browser to find any element in the DOM using its `id` and then change its properties or listen for events.

Example JavaScript Block (making our button work):

```
// 1. Find the HTML elements we need to work with
const missionButton = document.getElementById("complete-mission-btn");
const statusText = document.getElementById("status-text");

// 2. Define a function that runs when the button is clicked
function markMissionComplete() {
    // Change the text content of the paragraph
    statusText.textContent = "Status: COMPLETE";
    // Also change its style (CSS) to be green
    statusText.style.color = "green";

    // Log a message to the browser's developer console (F12)
    console.log("Mission status updated!");
}

// 3. Attach our function to the button's "click" event
missionButton.addEventListener("click", markMissionComplete);
```

This code finds the button and the paragraph from our HTML. When the button is clicked, it runs a function that changes the paragraph's text and its color. The `console.log` line is your single most useful tool for debugging—it prints messages to the browser's developer console (press F12 to open it) so you can see what your code is doing.

What to look for: Find event listeners (like `addEventListener("click", ...)`) to understand the application's core logic. This tells you what the tool actually *does*.

To learn more: To see these concepts in action, watch this JavaScript Crash Course (https://www.youtube.com/watch%3Fv%3DhdI2bqOjy3c)

HTML, CSS, JavaScript Explained [in 4 minutes for beginners] by Danielle Thé: http://www.youtube.com/watch?v=gT0Lh1eYk78

HTML, CSS, and Javascript in 30 minutes by devdojo: http://www.youtube.com/watch?v=_GTMOmRrqkU

Chapter 4: Your Project's Blueprint: File Structure

> **Concept: How you organize your files is critical. We start simple and add structure as needed.**

Imagine you're setting up a new workspace to build a project. At first, you might just have one toolbox where you keep everything—your hammer, screwdriver, and nails all live together. This is simple and works perfectly when you only have a few tools. This is your **file structure**: it's simply the way you decide to organize all the pieces of your project.

As your projects get bigger, that single toolbox gets crowded. It becomes hard to find the right screwdriver when it's buried under a pile of wrenches. To be more efficient, you might set up a proper workshop: a shelf for power tools, a drawer for hand tools, and labeled bins for screws and nails. While it takes a little more effort to set up, this organization saves you a massive amount of time in the long run.

Organizing a software project works exactly the same way. The two structures below aren't just for different project sizes; they're for different *purposes*: one for building and one for sharing.

1. The Single-File Start (For Simple Projects)

For your first few applications, the simplest approach is the best. All of your code—your HTML (structure), CSS (style), and JavaScript (action)—can live together inside a single file named `index.html`.

This method has powerful advantages. A single file is incredibly easy to manage and share. You can email it, put it on a shared drive, or carry it on a USB stick without worrying about missing pieces. When the user receives it, they only have to double-click that one file, and the application runs instantly because everything it needs is already inside.

Your `index.html` file would be structured like this:

```
<!DOCTYPE html>
<html>
<head>
   <title>My App</title>
   <!-- All your CSS rules go inside this style tag -->
   <style>
      /* ... Your CSS code here ... */
   </style>
</head>
<body>
   <!-- All your HTML structure goes here -->
   <h1>My Application</h1>

   <!-- All your JavaScript logic goes inside this script tag -->
```

```
    <script>
        // ... Your JavaScript code here ...
    </script>
</body>
</html>
```

When to use this: This is a great way to start and is perfect for small tools, calculators, and simple checklists where the total amount of code is manageable.

2. The Structured Folder (For Building and Development)

As your application grows, a single file can become crowded. The more professional and scalable approach is to use a structured folder. By separating your HTML, CSS, JavaScript, and data into different files and folders, you create a clean, organized workspace that is much easier to debug and update.

A standard, easy-to-understand structure looks like this:

```
my-app/
├── index.html
├── css/
│   └── style.css
├── js/
│   └── script.js
```

What each part does:

- **`my-app/`**: The main project folder that contains everything.
- **`index.html`**: The skeleton of your app. It will now simply point to your external CSS and JavaScript files.
- **`css/style.css`**: Holds all your styling rules. Your `index.html` will contain a line like `<link rel="stylesheet" href="css/style.css">`.
- **`js/script.js`**: Contains your application's logic. Your `index.html` will contain a line like `<script src="js/script.js"></script>`.

3. The Best of Both Worlds: The WCT Workflow

So which approach is better? The answer is **both**, used at different stages. You will use the **Structured Folder** while you are building the application, and the **Single File** as the final product you share.

This is the standard workflow for modern development and is the process facilitated by the **Warfighter Coder Tool (WCT)**, which we will discuss in a later section..

The workflow is simple:

1. **Develop:** You build your application using the clean, organized **Structured Folder**. This makes it easy to manage your code, find bugs, and add new features. Crucially, this separation also allows you to get more out of your AI pair programmer. Instead of asking the LLM to generate one massive, complex file, you can give it smaller, more focused tasks. For example, you can work on each file separately, asking the AI to "write the JavaScript logic in `script.js`" and then, in a separate conversation, "create the layout in `style.css`." This method keeps your instructions clear, helps the AI focus on one problem at a time, and results in better, more accurate code.
2. **Compile:** When you are ready to share your tool, you will use a feature in the WCT called the "Compiler." This tool reads your `index.html`, finds all the linked `style.css` and `script.js` files, and automatically bundles them all into a single, self-contained `index.html` file.

This process gives you the best of both worlds: the clean organization of a structured folder while you build, and the simple portability and security of a single file when you share.

Junior vs Senior React Folder Structure - How To Organize React Projects by Web Dev Simplified [http://www.youtube.com/watch?v=UUga4-z7b6s]

This Folder Structure Makes Me 100% More Productive by Web Dev Simplified [http://www.youtube.com/watch?v=xyxrB2Aa7KE]

How To Structure A Programming Project… by Tech With Tim: http://www.youtube.com/watch?v=CAeWjoP525M

Chapter 5: Understanding Data Formats: JSON and CSV

> **Concept: To get data in and out of your app, you need a standard format. Think of these as universal digital paperwork.**

Your application needs a way to save its state, whether that's a list of completed tasks or a set of coordinates. It also needs a way to share that data with other people or programs. To do this reliably, we use standardized formats that act like universal digital paperwork. Any program that understands the format can read the data, no matter who created it.

For our purposes, we will focus on two of the most common and useful formats: **JSON** for saving your application's memory and **CSV** for exporting data to be used in spreadsheets.

You will not be be manually creating files in these formats, but you will be asking your LLM coding partner to include import and export functions to create and handle files in these formats.

1. JSON (JavaScript Object Notation)

Think of JSON as a digital notecard or a labeled container for a single piece of information. It uses a simple `label: value` system to store data in a way that is easy for both humans and computers to read. The "label" is always a string in double quotes, followed by a colon, and then the "value."

JSON is the native language for storing structured data in JavaScript applications. It's perfect for saving the exact state of your app so you can load it back later exactly as it was.

Example: A Single JSON Object

Imagine you want to store information about a single task. In JSON, it would look like this, contained within curly braces `{}`.

```
{
  "Task": "Pre-flight check",
  "Status": "Complete",
  "Priority": 1,
  "IsUrgent": true
}
```

Example: A List of JSON Objects

To store a list of tasks, you would wrap multiple objects in square brackets `[]`, separating each object with a comma. This creates a list, or an "array," of tasks.

```
[
  {
```

```
  "Task": "Pre-flight check",
  "Status": "Complete"
 },
 {
  "Task": "Review mission brief",
  "Status": "In Progress"
 },
 {
  "Task": "Verify comms",
  "Status": "Not Started"
 }
]
```

When to use it: JSON is the primary format you will use for the **Import/Export** feature in your apps. When a user clicks "Save," you will bundle up your application's data into a JSON file for them to download. When they click "Load," you will read that same file back in to restore the state.

2. CSV (Comma-Separated Values)

Think of CSV as the simplest possible spreadsheet, stored in a plain text file. It's a universal format that almost every data analysis tool, especially Microsoft Excel, can open and understand instantly.

In a CSV file, every line is a **row**, and the values in that row (the **columns**) are separated by commas. The first line is typically used as a header row to label the columns.

Example: A Simple CSV File

If you wanted to export the same task list from above into a CSV file, it would look like this:

```
Task,Status,Priority,IsUrgent
Pre-flight check,Complete,1,true
Review mission brief,In Progress,2,true
Verify comms,Not Started,3,false
```

When you open this file in Excel, it will automatically be organized into a clean, four-column table.

When to use it: CSV is the perfect format when you need to get data *out* of your application and into a different program for analysis, reporting, or briefing. If a user needs to take the results from your tool and put them into a PowerPoint slide or an after-action report, providing an "Export to CSV" button is the most effective way to do it.

What Is JSON | Explained by Hostinger Academy [http://www.youtube.com/watch?v=cj3h3Fb10QY]

Understanding CSV Files by macmostvideo [http://www.youtube.com/watch?v=UofTplCVkYI]

Part 2: The Core Workflow

(Focus: A hands-on, step-by-step guide to defining a problem and building your first application.)

Chapter 6: The Golden Rule: OPSEC First

> **Concept: Public AI models train on your input. Never, ever use real, sensitive, or classified data when prompting them.**

******* Please read through this chapter before building.*******

“OPSEC First” is the single rule that, if broken, can have the most severe consequences. When you interact with a publicly available Large Language Model (LLM)—like the ones on the open internet—you must assume that your conversation is being recorded and used to train future versions of the AI.

Think of it like having a conversation in a public square. You have no expectation of privacy, and you never know who might be listening or what they will do with the information they overhear. Pasting sensitive information into a public LLM is the digital equivalent of shouting classified details in that square. The data leaves your control forever and becomes part of the model's vast knowledge base, creating a permanent data spill that cannot be undone.

For this reason, maintaining Operational Security (OPSEC) is not just a best practice; it is your primary responsibility as a warfighter-coder.

What NEVER to Paste into a Public AI

Your guiding principle should be to treat any public AI (commercially offered for personal use) as an unclassified, public forum. Before you paste any text, ask yourself: "Would I be comfortable if this information appeared on the front page of a newspaper?" If the answer is no, do not paste it.

This includes, but is not limited to:

- **Classified Information:** Any data marked as CONFIDENTIAL, SECRET, or TOP SECRET.
- **Controlled Unclassified Information (CUI)** and For Official Use Only (FOUO).
- **Personally Identifiable Information (PII):** Real names, social security numbers, phone numbers, addresses, etc.
- **Operational Details:** Real unit names, ship names, tail numbers, mission IDs, locations, dates, or exercise names.
- **System or Network Information:** IP addresses, server names, credentials, or internal configuration details.

How to Sanitize Data for the AI

Your LLM often needs to understand the *structure* of your data to help you write code that can process it. You can provide this structure without revealing the sensitive contents by creating generic, unclassified samples. This process is called **sanitizing**.

The goal is to replace the real data with plausible but fake placeholders, while keeping the format (the labels and data types) exactly the same.

Bad Example (NEVER do this): You want to build a tool to track mission readiness. Your data looks like this, and you paste it directly into the AI.

```
{
  "vessel_name": "USS NEVERSAIL",
  "mission_id": "123-ABC-789",
  "crew_chief": "PO1 Smith, John"
}
```

This is a serious OPSEC violation. You have exposed a real (or realistic) ship name, a mission identifier, and the name of a crew member.

Good Example (Sanitized): Instead, you replace the sensitive values with generic placeholders before pasting.

```
{
  "unit_name": "Unit A",
  "task_id": "Task-001",
  "poc_name": "Person 1"
}
```

This sanitized version gives the AI everything it needs to know. It understands that `unit_name` is a string, `task_id` is an identifier, and `poc_name` is a person's name. It can now write code to parse and display this structure for you, without you ever having exposed sensitive information. Always sanitize your data before you ask for help.

Learn OpSec Fundamentals in 15 Minutes! by Sam Bent: http://www.youtube.com/watch?v=rMSgnOYcEVE

Chapter 7: Finding the Real Problem (The TAG Framework & JTBD)

> **Concept: Before you write a single prompt, you must define the mission. A great tool solves a real, validated problem.**

In the military, no operation begins without a clear mission objective. You don't just "go east"; you conduct a reconnaissance patrol to identify enemy positions at a specific grid coordinate to inform the commander's next move. The same discipline applies to building software. The most common reason a software project fails is not because the code is bad, but because it solves the wrong problem.

An LLM is a powerful tool, but it lacks your context. It doesn't know what it's like to stand a watch, conduct a pre-flight check, or manage maintenance logs. The quality of the solution it generates is entirely dependent on the quality of the mission you give it. This chapter will teach you how to define that mission with precision using two powerful frameworks: the **TAG Framework** for defining the problem, and **Jobs-to-be-Done (JTBD)** for understanding the user's true motivation.

1. Frame the Problem with TAG (Target, Actual, Goal)

The TAG framework is a simple but powerful tool for stating a problem in plain language. It forces you to be specific about who is affected, what the current pain point is, and what a better future looks like.

- **T - Target:** Who is the user? Be specific. "A watchstander" is okay, but "An E-4 Quartermaster standing bridge watch on a DDG" is much better because it gives you a concrete person to visualize.
- **A - Actual:** What is the current, concrete reality? Describe the process that is inefficient, error-prone, or frustrating. This is the "pain" you are trying to solve.
- **G - Goal:** What is the desired outcome? This is a description of a better future, *not* a description of an app. Don't say "The goal is to have a digital checklist app." Instead, say "The goal is for sailors to capture results accurately without manual re-typing."

Example TAG Statement:

- **Target:** Junior sailors in the Deck Department who are tasked with weekly maintenance checks.
- **Actual:** They currently use laminated paper cards and a grease pencil. At the end of the day, a Petty Officer spends 45 minutes manually collecting these cards and typing the results into a maintenance log. The cards get smudged, are hard to read, and this process often leads to transcription errors.

- **Goal:** The sailors can complete their checks quickly, and the final results are captured accurately in a digital format, eliminating manual transcription and freeing up the Petty Officer for more important leadership tasks.

2. Go Deeper with "Jobs-to-be-Done" (JTBD)

Now that you have a clear problem statement, you need to understand the underlying motivation. The "Jobs-to-be-Done" framework helps you look past superficial requests ("I need an app") and understand the real "job" the user is trying to accomplish. Users don't "hire" a product just to have a new tool; they "hire" it to make progress in their life.

In our TAG example, the "job" isn't "to use a digital checklist." The real jobs are:

- **For the Junior Sailor:** "Help me quickly and accurately capture my maintenance findings so I can finish my work and know it was done right."
- **For the Petty Officer:** "Help me effortlessly gather and verify my team's maintenance results so I can trust our readiness and spend less time on paperwork."

Understanding the JTBD is your secret weapon. It focuses you on the user's goal, not on specific features. To uncover the real job, you need to talk to your target user. This is a discovery interview, not a sales pitch. Your goal is to listen, not to convince.

User Interview DOs and DON'Ts

DO Ask This...	DON'T Ask This...
"Tell me about the last time you..."	"Would you use an app that..."
"Walk me through your current process."	"Don't you think it would be better if..."
"What's the hardest part of...?"	"Do you like this idea I have?"
"How are you solving this now?"	"What features do you want?"

Listen for frustrations, workarounds, and wasted time. That's where the real opportunity is.

3. The AI-Assisted Workflow

Once you have notes from your user interview, you can use your LLM as a creative partner to help you structure your thoughts and plan your project.

Step 1: Extract Jobs & Tool Ideas

Feed your interview summary and your TAG statement to the AI. Ask it to perform the same analysis we just did: identify the core "Jobs-to-be-Done" and then brainstorm a few different tool

ideas that could solve that job. This helps you think broadly before committing to a single solution.

Step 2: Define Your Minimum Viable Product (MVP)

Once you've chosen a single job and a single tool idea, the next step is to define the smallest possible version you can build to see if your idea actually works. This is your **Minimum Viable Product (MVP)**. It's not your final product; it's an experiment designed to test a hypothesis.

You can ask your AI to generate an "MVP Ladder"—a list of 3-5 versions of your tool, starting with the absolute simplest version and gradually adding features. This is an incredibly powerful technique for scoping your project and preventing you from trying to build everything at once. Your goal is to build MVP 1, test it with your user, and learn from their feedback before you even think about building MVP 2. This iterative process of building, testing, and learning is the heart of effective software development.

The ultimate guide to JTBD | Bob Moesta (co-creator of the framework) by Lenny's Podcast:
http://www.youtube.com/watch?v=xQV7HVyAJjc

Clay Christensen: The Jobs to be Done Theory by HubSpot Marketing:
http://www.youtube.com/watch?v=Stc0beAxavY

Chapter 8: How to Talk to Your AI (The TCF Prompt Pattern)

> **Concept: Use the insights from your MVP ladder to craft a precise build instruction for the AI. A clear prompt is the difference between a working app and a frustrating mess.**

You've defined your mission (Chapter 7), you know the basic languages (Chapter 3) and file structures (Chapter 4), and you've established your security posture (Chapter 6). Now it is time to translate your mission objective into a clear, actionable command for your AI pair programmer. A vague request like "make me a checklist app" will produce a generic and likely unusable result. A precise, structured prompt will give you a strong foundation to build upon.

The most effective way to structure your request is with the **TCF Prompt Pattern**. This simple framework ensures you provide the AI with everything it needs to know to give you a useful first draft.

- **T is for Task:** State exactly what you want to build. Be specific, clear, and concise. This comes directly from the MVP you scoped in the previous chapter. Describe the core function of the tool.
    - *Bad:* "Make a checklist."
    - *Good:* "Build a simple checklist application for tracking daily pre-flight inspections for five specific items."
- **C is for Context:** Explain who the user is and what the operational constraints are. This is the most critical part of the prompt, as it guides the AI's architectural choices. If you don't provide context, the AI will make assumptions that are likely wrong for our environment.
    - *Example:* "The user is an aircraft maintainer on a locked-down government laptop with no internet access. The tool must be fully functional when opened as a local `file:///` and cannot make any network requests. The user is not a technical expert, so the interface must be simple and intuitive."
- **F is for Format:** Tell the AI exactly how you want the code delivered. This is where you apply your knowledge of file structures from Chapter 4. Your choice of format depends on the complexity of your application and is key to managing both your project and your AI partner effectively.

Workflow 1: The Single-File Build for Simple Apps

For your first MVP or for very simple, single-purpose tools, asking for everything in one file is the fastest way to get a working prototype. It minimizes complexity and gives you a single, portable file to test immediately.

When to Use This:

- You are building the first version (MVP 1) of an idea.
- The tool is very simple (e.g., a basic calculator, a single checklist).
- The total code is likely to be less than a few hundred lines.

How to Prompt for a Single File:

You combine the T, C, and F into one comprehensive prompt. Your "Format" instruction is direct and explicit.

Example TCF Prompt (Single-File Checklist App):

> **T (Task):** Create a single-file HTML checklist app. The list should contain the following five pre-flight tasks: "Check tire pressure", "Inspect landing gear", "Verify fuel levels", "Test communication systems", and "Review flight plan".
>
> **C (Context):** This app will be used offline by a pilot on a government laptop with no internet. It must run from a single local HTML file.
>
> **F (Format):** Provide all the code in a **single HTML file**. All CSS and JavaScript must be included internally within `<style>` and `<script>` tags. Do not use any external CDN links for libraries or fonts.

When you give this prompt to your AI, it will generate one complete block of code that you can save directly into your `index.html` file and run immediately.

Workflow 2: The Multi-File Build for Complex Apps

As your application grows, a single file becomes difficult to manage. More importantly, LLMs produce better, more focused results when you ask them to build one component at a time. By breaking the problem down, you act as a better manager, guiding the AI to create a clean, organized, and ultimately more successful project.

When to Use This:

- Your application has multiple features or a more complex user interface.
- You are building beyond the initial MVP and adding significant new functionality.
- You want a project structure that is easier to debug and maintain.

How to Prompt for Multiple Files:

You will act as a project manager, having a conversation with your AI to build the application piece by piece.

Step 1: Create Your Folder Structure

Before you even start prompting, create the folders you'll need on your computer, as described in Chapter 4:

```
my-complex-app/
├── css/
├── js/
└── data/
```

Step 2: Prompt for `index.html` (The Blueprint)

Your first prompt will ask the AI to create the main HTML file, but instead of including the CSS and JavaScript, you'll tell it to *link* to the files you just created placeholders for.

Example TCF Prompt (`index.html`):

> **T (Task):** Create the main HTML structure for a multi-mission tracking application. It needs a header, a main content area for a list of missions, and a footer.
>
> **C (Context):** This is the main HTML file for an offline application.
>
> **F (Format):** Provide only the `index.html` file. Link to an external stylesheet at `css/style.css` and an external JavaScript file at `js/script.js`. Do not include any CSS or JavaScript in this file.

Save the AI's response as `index.html` in your `my-complex-app/` folder.

Step 3: Prompt for `style.css` (The Style)

Now, in a new conversation with your AI (or by providing the HTML as context), ask it to generate the styles.

Example TCF Prompt (`style.css`):

> **T (Task):** Now, create the CSS for the application. Give it a clean, professional dark-theme design suitable for a tactical environment.
>
> **C (Context):** This CSS will be used for the `index.html` file I just created.
>
> **F (Format):** Provide only the CSS code for the `style.css` file.

Save this response inside your `css/` folder as `style.css`.

Step 4: Prompt for `script.js` (The Action)

Finally, ask for the application logic.

Example TCF Prompt (`script.js`):

> **T (Task):** Now, write the JavaScript logic for the application. For now, just make the "Add Mission" button log a message to the console when clicked.
>
> **C (Context):** This JavaScript will be used for the `index.html` file. The button has an id of `add-mission-btn`.
>
> **F (Format):** Provide only the JavaScript code for the `script.js` file.

Save this response inside your `js/` folder as `script.js`.

By following this multi-file workflow, you have guided the AI to build a clean, structured application. This method is more deliberate, but it is the key to building larger, more capable tools and getting the highest quality output from your AI partner.

Master the Perfect ChatGPT Prompt Formula (in just 8 minutes)! by Jeff Su:
http://www.youtube.com/watch?v=jC4v5AS4RIM

Part 3: The Build Process

Chapter 9: Your First Build: The Checklist App

> **Concept: Turn your prompt into a working file by pasting it into your AI and saving the response.**

This is the moment where all your planning turns into something tangible. You have a clear mission objective, a precise prompt, and an understanding of the basic components. Now, you will execute the core task of AI-assisted development: commanding your AI to generate the code and then saving that code as a functional application. We will use the simple, single-file workflow to build our first checklist app.

Step 1: Prepare Your Workspace

First, create a new folder on your computer to hold the project. This keeps your work organized. A clear name is best. For this example, let's call it `checklist-app`. This folder is the home for your new tool.

Step 2: Generate the Code

Open your command-approved AI chatbot (such as NIPRGPT, ChatGPT, etc.). This is the interface where you will give your instructions to your AI pair programmer.

Now, copy the complete TCF prompt you crafted in the previous chapter. This prompt contains the task, context, and format—everything the AI needs to generate a useful first draft. For our checklist app, the prompt is:

> **T (Task):** Create a single-file HTML checklist app. The list should contain the following five pre-flight tasks: "Check tire pressure", "Inspect landing gear", "Verify fuel levels", "Test communication systems", and "Review flight plan".
>
> **C (Context):** This app will be used offline by a pilot on a government laptop with no internet. It must run from a single local HTML file.
>
> **F (Format):** Provide all the code in a **single HTML file**. All CSS and JavaScript must be included internally within `<style>` and `<script>` tags. Do not use any external CDN links for libraries or fonts.

Paste this entire prompt into the AI's chat window and send it. The AI will process your request and, within a few seconds, generate a complete block of code.

The response should be a single, large block of text starting with `<!DOCTYPE html>` and ending with `</html>`. Inside, you should see the `<style>` and `<script>` tags that contain all the necessary CSS and JavaScript, just as you requested. Most AI tools provide a "Copy"

button to grab the entire response. Use it, or manually select all the code to ensure you get every line.

Step 3: Save Your Application File

With the code copied to your clipboard, open your text editor (like Notepad++ or any other plain text editor). Create a new, blank file. Notepad works great for this.

Paste the entire block of code you copied from the AI into this empty file.

Now, go to "File" -> "Save As...". Navigate into the `checklist-app` folder you created in Step 1. For the filename, type **`index.html`**. The `.html` part is critical—it's the file extension that tells your computer this is a web page that should be opened by a browser. **Ensure the "Save as type" is set to "All Files" to prevent your editor from adding an extra `.txt` at the end.**

Click "Save."

Step 4: Run Your First Application

You have now created your first working application. To run it, navigate to your `checklist-app` folder using your computer's file explorer. You should see your `index.html` file.

Double-click the `index.html` file.

It will open directly in your default web browser (like Edge or Chrome). You should see your checklist application on the screen: a simple, functional tool built from a single prompt.

You have successfully translated a mission need into a working piece of software. It might not be perfect yet, but it's a real, tangible starting point. The next chapter will teach you the most important part of the process: how to test, debug, and refine your application until it's mission-ready.

How To Create To-Do List App Using HTML CSS And JavaScript by GreatStack: http://www.youtube.com/watch?v=G0jO8kUrg-I

Chapter 10: The Build Loop: Test, Triage, and Refine

> **Concept: Your first version won't be perfect. The core skill is iterating quickly using standard browser tools.**

Your AI-generated code is a first draft, not a finished product. It will almost certainly have bugs, and it might misunderstand parts of your request. This is not a failure; it is an expected and normal part of the development process. No coder, human or AI, gets it right on the first try.

Your job now is to become a skilled manager and quality assurance tester. By finding the flaws and providing clear, specific feedback to your AI, you can guide it to a correct solution. This iterative cycle of **Test -> Triage -> Refine** is the **Build Loop**, and mastering it is the most important skill in AI-assisted development.

Step 1: Test Your Application

The first step is simple: use your tool. Navigate to your project folder and double-click the `index.html` file to open it in your web browser.

Once it's open, interact with every part of it. Click every button. Check every box. Type into every field. Ask yourself:

- Does it look right? Is anything misaligned or styled incorrectly?
- Does it function as expected? When you click a button, does it do what you intended?
- Does anything break? Does the application become unresponsive or behave strangely after a certain action?

Take note of anything that seems wrong. Your personal experience as the user is the first and most important form of testing.

Step 2: Triage (F12 is Your Best Friend)

When something doesn't work, your most powerful ally is the **Developer Console** built into every modern web browser. It's a window that gives you a behind-the-scenes look at your application's health.

Press the F12 key on your keyboard to open it.

Inside this new window, look for a tab named **"Console."** If there are any errors in your application's code, they will almost always appear here, usually highlighted in red.

These error messages are gold. They are not a sign of failure; they are a precise report from the field, telling you exactly what went wrong and often on which line of your code the problem occurred. An error like `Uncaught ReferenceError: missionButton is not defined at index.html:42` tells you that on line 42, your JavaScript tried to use a variable named

`missionButton` that it couldn't find. This is the exact, specific feedback your AI needs to fix the problem.

Step 3: Refine and Re-Prompt

Now you will guide your AI to fix the bug. Do not just tell the AI "it's broken." Provide it with the specific intelligence it needs to succeed.

1. **Copy the exact error message** from the F12 console.
2. **Copy the full code** from your `index.html` file.
3. **Paste both** into your AI chatbot with a clear, direct request for a fix. Structure your prompt like this:

   "The code you provided is producing an error. Please find the bug and provide the corrected, full code.

   Error Message: `[Paste the exact error message from the F12 console here]`

   My Full Code: `[Paste your complete index.html code here]`"

The AI will analyze the error in the context of your code and generate a new, corrected version. It might explain the fix, but your primary goal is the updated code block.

Step 4: Repeat Until It Works

Replace the old code in your `index.html` file with the AI's new version, save the file, and go back to your web browser. **Refresh the page (F5 or Ctrl+R)** to load the new code.

Now, repeat the loop: **Test** the application again. If it works, great. If you see a new error, **Triage** it in the F12 console and **Refine** your prompt to the AI.

This cycle of `Test -> Triage -> Refine` is the fundamental rhythm of building software with an AI. By iterating quickly, you can take a buggy first draft and shape it into a reliable, mission-ready tool.

Debugging JavaScript - Chrome DevTools 101 by Chrome for Developers:
http://www.youtube.com/watch?v=H0XScE08hy8

Chapter 11: Making Your App Remember: `localStorage`

> **Concept: Your app needs a "file cabinet" to save its work between sessions. `localStorage` is a simple storage system built into every browser for exactly this purpose.**

You have successfully built and tested your first application. It looks right, it functions correctly, but it has a major flaw: it has amnesia. Every time you close or refresh the page, all your progress vanishes. The checklist resets, and any data you've entered disappears. This happens because the application's state—the current status of all its variables and user inputs—only exists in the computer's active memory. When the page closes, that memory is wiped clean.

To create a truly useful tool, your application needs a way to remember things. It needs a memory. For simple, offline applications, the easiest and most effective way to achieve this is by using a browser feature called `localStorage`.

What is `localStorage`?

Think of `localStorage` as a small, digital file cabinet that the browser provides for your application. It is a secure and private storage space that is tied directly to the file you are running.

- **It's Persistent:** Data saved in `localStorage` stays there even after you close the browser tab or shut down your computer. When you open the `index.html` file again, the data will still be there, ready to be loaded.
- **It's Private:** The `localStorage` for your `checklist-app/index.html` is completely separate from the storage of any other website or local file. No other application can see or interfere with its data.
- **It's Simple:** It is designed for storing small amounts of text-based data, making it perfect for things like user settings, form data, or the state of a checklist.

How It Works: Key-Value Pairs

`localStorage` works like a simple dictionary or a set of notecards. You store information in **key-value pairs**.

- The **key** is a unique name you give to a piece of data, like a label on a folder (e.g., `"checklistState"`).
- The **value** is the actual data you want to store, which must be text (e.g., `["true", "false", "true"]`).

Your JavaScript code will have two main jobs: when the user makes a change, it will **save** the current state of the checklist into `localStorage` under a specific key. When the page first

loads, it will **load** whatever data is stored under that key to restore the checklist to its previous state.

Workflow: Iterating on Your App to Add Memory

You will use the same **Build Loop** (`Test -> Triage -> Refine`) from the previous chapter to add this new feature. You don't need to know how to write the `localStorage` code yourself; you just need to clearly describe your goal to the AI.

Step 1: Craft an Update Prompt

Your prompt needs to be specific and provide the AI with all the necessary context. The most effective way to do this is to give the AI the complete, working code from your current `index.html` file and tell it what you want to add.

1. **Copy the entire code** from your working `index.html` file.
2. **Paste it into your AI chatbot** with a clear request for the update.

Example Update Prompt:

> "Here is the code for my working checklist app. Please update it to save the checklist's state to `localStorage`. When I close or refresh the page, the checked items should be remembered and reloaded.
>
> **My Full Code:** `[Paste your complete index.html code here]`"

This prompt is effective because it gives the AI the full context of what it's modifying and a clear, singular goal.

Step 2: Save and Test the New Version

The AI will return a new, complete version of your `index.html` file. The JavaScript inside the `<script>` tag will now be updated with the logic to save and load data.

1. **Replace the old code** in your `index.html` file with the new code from the AI.
2. **Save the file** and open it in your web browser.
3. **Test the new feature:** Check a few boxes on your list.
4. **Refresh the page** (you can press F5 or Ctrl+R).

If the AI did its job correctly, the boxes you checked should *still* be checked after the page reloads. Your application now has memory.

If it doesn't work, don't worry. This is part of the process. Go back to **Chapter 10** and follow the **Build Loop**: open the F12 console, look for errors, and use that information to craft a refinement prompt for your AI until the feature works perfectly.

What to Look For in the Code

As you build more apps, you'll start to recognize the patterns for `localStorage`. The two key functions your AI will use are:

- `localStorage.setItem(key, value)`: This is the "save" command. It tells the browser to store the `value` under the name you provide in the `key`. For example: `localStorage.setItem('myChecklist', '["true", "false"]');`
- `localStorage.getItem(key)`: This is the "load" command. It retrieves the data that was saved under that `key`.

Because `localStorage` can only store text, your AI will often use `JSON.stringify()` to convert your application's data (like a list of true/false values) into a text string before saving it, and `JSON.parse()` to convert it back into a usable format after loading it.

How to Use Local Storage in JavaScript by dcode: http://www.youtube.com/watch?v=k8yJCeuP6I8

JavaScript Cookies vs Local Storage vs Session Storage by Web Dev Simplified: http://www.youtube.com/watch?v=GihQAC1I39Q

Chapter 12: The Data Lifeline: Import & Export

> **Concept: Every offline app needs a way to get data in and out. Import and Export buttons are the lifeline that lets users back up and share their work.**

In the last chapter, you gave your application a memory using `localStorage`. This is a powerful step—it allows a single user on a single machine to pick up their work right where they left off. However, that "file cabinet" is bolted to the floor of that one browser. What happens if you switch computers? What if your browser's data gets cleared by a system update? What if you need to send your completed checklist to your supervisor?

Without a way to move data in and out of the application, your information is trapped. This is where the **Import/Export lifeline** comes in. Adding "Export" and "Import" buttons gives your users complete control over their data, making your simple offline tool more robust and useful.

The Two Functions of the Lifeline

1. **Export: Creating a Digital Backup**
   An "Export" button tells the application to take a snapshot of its current state—all the checked boxes, all the notes, everything that's been saved to `localStorage`—and bundle it into a downloadable file. This file acts as a permanent, portable backup that the user can save anywhere they like. It is the digital equivalent of making a photocopy of your important paperwork.
2. **Import: Restoring from a Backup**
   An "Import" button does the reverse. It allows a user to select a previously exported file from their computer and load its contents back into the application. This action overwrites the app's current state with the data from the file, instantly restoring it to that saved point in time. This is how a user can move their work from one machine to another or load a checklist file sent to them by a teammate.

Workflow: Adding the Lifeline to Your App

Just like adding `localStorage`, you don't need to know the specific code to implement this feature. You will use the same iterative Build Loop, providing your AI partner with a clear prompt and your current code.

For data transfer, we can use the two formats we learned about in Chapter 5: **JSON** for perfect state backup and **CSV** for easy use in spreadsheets.

Option 1: Prompting for JSON (Full App Backup)

This is the most common and important type of data lifeline. It saves and loads the application's exact state.

1. **Copy the entire code** from your working `index.html` file (the one that already uses `localStorage`).
2. **Paste it into your AI chatbot** with a clear, specific request.

Example Update Prompt (JSON):

> "Here is the code for my checklist app, which already uses `localStorage`. Please add an 'Export Data' button and an 'Import Data' button.
>
> 1. When the 'Export Data' button is clicked, the app should save all the current checklist data into a JSON file named `checklist-backup.json` and download it.
> 2. When the 'Import Data' button is clicked, it should allow the user to select a JSON file. The data from that file should then overwrite the current checklist state and also update `localStorage` and display the file name.
>
> **My Full Code:** `[Paste your complete index.html code here]`"

Option 2: Prompting for CSV (Export for Spreadsheets)

Add this feature for users to use your app's data in another program, like Excel.

Example Update Prompt (CSV):

> "Using the same code, please also add an 'Export to CSV' button. When clicked, this button should convert the current checklist data into a CSV format and download it as `checklist-export.csv`."

Testing Your New Lifeline

With the updated code, save it to your `index.html` and open it in the browser for testing:

1. **Make some changes:** Check a few boxes or enter some text into your app.
2. **Click the "Export Data" button.** A file (e.g., `checklist-backup.json`) should be saved to your computer's Downloads folder.
3. **Open the file in a text editor.** Do the contents look accurate for the state of your app?
4. **Make more changes in the app:** Check a different set of boxes.
5. **Click the "Import Data" button.** Select the saved `checklist-backup.json` file.
6. **Verify the result:** The app should revert to the state it was in when you clicked "Export."

If anything goes wrong, use the F12 Developer Console to find error messages and refine your prompt with the AI until the feature is working perfectly. By adding this data lifeline, you have transformed your simple tool into a resilient and truly practical piece of software.

The Easiest Way to Export to CSV File in JavaScript by dcode: http://www.youtube.com/watch?v=kGCGxRqZ6XI

Part 4: Hardening and Shipping Your Tool

(Focus: The professional steps to make your tool secure, compliant, and ready for real-world use.)

Chapter 13: Proving Your App is Secure (The Code Audit)

> **The Question: "How can I be 100% certain this application is truly offline and isn't sending my data to the internet?"**

This is the single most important question you must be able to answer before you share your tool with anyone. In an environment where data security is paramount, trust is not enough—you need proof. Fortunately, for the simple, offline applications we are building, providing definitive proof is straightforward.

The proof that your application is secure is based on a powerful and simple principle: **the code fundamentally lacks the necessary tools to send your information to the internet.**

A web page cannot communicate with an external server by accident. It requires specific, explicit JavaScript functions or HTML tags to send data from your browser to a remote location. By conducting a simple "Code Audit," we can scan our entire application and prove that these functions are completely absent.

The Tools a Browser Uses to "Talk" to the Internet

For a web page to send information out, it must use one of the following standard mechanisms. Your job is to search for these patterns in your `index.html` file.

1. The `fetch()` API

- **What it is:** This is the modern and most common way to make network requests. The name is self-explanatory: it "fetches" data from or sends data to a server.
- **What to look for:** Search your code for the word `fetch(`. A line of code that sends data would look something like this: `fetch('https://some-server.com/data', { method: 'POST', ... });`
- The `POST` method is the key indicator that data is being *sent*.

2. `XMLHttpRequest` (XHR)

- **What it is:** This is the older method for making network requests, but it is still widely used.
- **What to look for:** Search your code for `XMLHttpRequest`. The code would involve creating a new object and using its `.send()` method to transmit data.

3. HTML Form Submission

- **What it is:** A standard HTML `<form>` tag can be used to send data to a server without any JavaScript.
- **What to look for:** Search your code for `<form`. A form that sends data will have an `action` attribute that points to a server URL (e.g., `action="https://some-server.com/submit"`) and usually a `method="POST"` attribute.

If your code doesn't use these, it structurally cannot send your data to an external server.

The Proof: A Step-by-Step Code Audit

You don't need to be a programmer to perform this audit. You just need to know how to use the "Find" feature in your text editor.

1. **Open Your Final File:** Open your single, compiled `index.html` file in your text editor (like Notepad++). This is the exact file you intend to share.
2. **Search for Network Keywords (Ctrl+F):** Use the Find feature (usually Ctrl+F or Cmd+F) to search for the following terms, one by one:
    - `fetch(`
    - `XMLHttpRequest`
    - `action="http` (This specifically looks for form actions pointing to the internet).
    - `WebSocket(` (Used for real-time connections, which our apps don't use).

Expected Result: In our offline applications, these searches should return **zero results**. The code to perform these actions simply doesn't exist in the file.

What About `<script>` and `<link>` Tags?

You might notice that the AI sometimes includes lines like `<script src="https://cdn.tailwindcss.com"></script>` or `<link href="https://some-font-library.com">` in early drafts. It's critical to understand the difference between these and the data-sending functions above.

These tags make a **read-only request**. The browser is asking the server, "Can I *get* this file?" This is a `GET` request, equivalent to downloading a file. It is fundamentally different from a `POST` request, which says, "Please *take* this data I am sending you."

While our final, compiled `index.html` file will have these scripts and styles embedded directly (as taught in Chapter 8) and will therefore make no external requests at all, even a development version with these links is not designed to *send* your application's data anywhere.

Conclusion of the Proof

By performing this simple audit, you can prove with certainty that your application is secure for offline use.

1. **No Data Sending Functions:** The code does not contain any `fetch`, `XMLHttpRequest`, or `WebSocket` calls necessary to transmit your data.
2. **No External Form Submissions:** Any `<form>` elements in the app are handled locally by JavaScript and lack the `action` attribute required to send data to a server.
3. **Read-Only Asset Loading (if any):** Any external links are for downloading visual components only and do not send any of your personal information.

Therefore, you can conclude with certainty that **no code in this application provides any of your application's data to the internet.** Its security is structurally guaranteed by the limitations of the code itself.

7 Security Risks and Hacking Stories for Web Developers by Fireship:
http://www.youtube.com/watch?v=4YOpIL_i9Oxs

Chapter 14: Preparing for Approval: Your Documentation Packet

> **The Goal: Make it easy for your local cyber/IA team to say "yes."**

You've built a functional tool, proven it's secure, and added the necessary features to make it useful. The final hurdle before you can share your application is getting it approved for use on the network. This process can seem daunting, but for the simple, offline tools we are building, it can be straightforward if you do one thing: **make the security team's job easy.**

Your local Information Assurance (IA) or Cybersecurity team is looking for risk. Their job is to verify that your application is safe and won't introduce vulnerabilities. Instead of waiting for them to audit your code from scratch, you can anticipate their questions and provide all the answers upfront in a clear, organized **Documentation Packet**. By doing their homework for them, you build trust and dramatically accelerate the approval process.

What Your Approver Needs to Know

The security team will ask a few key questions:

1. **What does it do?** (Purpose and function)
2. **Does it talk to the internet?** (We proved it doesn't in Chapter 13)
3. **How does it handle data?** (Input, storage, and output)
4. **Is it built with any third-party code?** (Libraries and dependencies)
5. **How do you know it works?** (Basic testing and verification)

Your documentation packet will answer every one of these questions clearly and concisely. You will create a new folder named `docs/` inside your main project folder to hold these files.

The Four Key Documents in Your Packet:

1. **`README.md` (The "What"):** This is a simple, plain-language summary of your application. It should explain what problem the tool solves, who it's for, and how to run it (e.g., "Double-click the `index.html` file").
2. **`SECURITY-NOTE.md` (The "How"):** This is the most important document. It explicitly states the security posture of your app. Here, you will formalize the findings from your code audit in Chapter 13. Your note should state clearly:
    - "This is a purely static HTML/CSS/JS application designed for offline use."
    - "It makes **no external network calls**."
    - "All data is stored locally in the user's browser via `localStorage` and can be deleted by the user."
    - "All third-party libraries (if any) are vendored locally."
3. **`SOFTWARE-BOM.txt` (The "With What"):** BOM stands for "Bill of Materials." This is a simple text file that lists all the third-party code your application uses. If your AI used a

library like Chart.js for a graph, you would list it here along with its version number and license (e.g., "Chart.js | v4.4.1 | MIT License"). Your AI can provide this information for you.

4. **`VERIFICATION.md` (The "Proof"):** This is a simple, manual test script that proves you have tested the tool. It's a checklist of steps that anyone can follow to verify that the application works as expected.

Workflow: Using Your AI to Generate the Documentation

Just as you used the AI to write the code, you can use it to write the documentation. The key is to provide the AI with your full, final application code and a specific prompt asking it to analyze the code against official security controls.

Step 1: Consolidate Your Code

If you built your app using a multi-file structure, use your LLM Formatter or a similar tool to combine your `index.html`, `style.css`, and `script.js` into a single block of text.

Step 2: Craft a Security Report Prompt

Your prompt should ask the AI to act as a security reviewer and assess your application against a known standard. For our purposes, the **Application Security and Development (ASD) STIG** is the relevant guide.

Example Prompt to Generate Your Documentation:

> "You are a security reviewer. Audit the provided offline, static HTML/CSS/JS codebase against the Application Security & Development STIG controls listed below.
>
> **Controls to Assess:** V-222388 (Clear temp storage), V-222602 (Protect from XSS), V-222642 (No embedded secrets), V-222665 (No unapproved mobile code).
>
> For each control, determine if it is **Applicable**, and if so, assess the code's compliance status as **Compliant** or **Non-Compliant**, providing evidence from the code.
>
> Based on your audit, generate the content for a `README.md` and a `SECURITY-NOTE.md` for this application.
>
> **My Full Code:** `[Paste your complete application code here]`"

Step 3: Review and Save the AI's Output

The AI will generate the text for your documentation files. Your job is to:

1. **Read and Verify:** Read through the AI's analysis. Does it make sense? Does it accurately reflect what your code does? The AI is your assistant, but you are the final authority.
2. **Save the Files:** Copy the generated content for the `README.md` and save it to `docs/README.md`. Do the same for `SECURITY-NOTE.md`.
3. **Create the BOM and Verification:** Ask the AI to list the libraries for your `SOFTWARE-BOM.txt` and to generate a simple test plan for your `VERIFICATION.md`.

By preparing this packet, you are demonstrating professionalism and a commitment to security. You are not just handing over a tool; you are delivering a complete, documented, and verifiable solution that makes it easy for your leadership to say "yes."

HOW TO EASILY WRITE SOFTWARE REQUIREMENTS SPECIFICATION by Jelvix | TECH IN 5 MINUTES: http://www.youtube.com/watch?v=PtJmjPkrSUE

Chapter 15: From Folder to Single File for Distribution

> **Concept: For easy and secure distribution, you will bundle your entire multi-file project into a single, self-contained `.html` file that anyone can run.**

Throughout this guide, we've followed a professional workflow: developing our application in a clean, structured folder with separate files for HTML, CSS, and JavaScript. This organization is essential for building and debugging effectively. However, sharing a folder full of files can be cumbersome and error-prone. A user might forget to download a critical file, or a security scanner might flag a folder containing multiple script files.

The solution is to perform a final "compilation" step. This process takes all the individual pieces from your development folder and bundles them into one master `index.html` file. This single file is the ultimate in portability: it's easy to share, impossible to misconfigure, and simple for anyone to use. They just double-click it.

This chapter will teach you how to instruct your AI to perform this compilation, turning your organized project into a distributable tool.

The "Compilation" Workflow

This is the last step you take after your application is fully built, tested, and ready to be shared. You will provide your AI with the contents of all your project files and ask it to combine them.

Workflow: Using Your AI to Compile Your Project

Step 1: Gather Your Code

Open each of the files from your structured project folder in your text editor:

- `index.html`
- `css/style.css`
- `js/script.js`
- Any other CSS or JS files you may have added.

You will be copying the full contents of each of these files.

Step 2: Craft a Consolidation Prompt

Your prompt will be a multi-part instruction. You will provide the AI with the content of each file and tell it exactly how to combine them.

Example Consolidation Prompt:

> "You are a web developer tasked with compiling a multi-file project into a single, self-contained HTML file for offline distribution.

Task: Take the following HTML, CSS, and JavaScript files and combine them.

1. Take the content from the CSS file and place it inside a `<style>` tag in the `<head>` section of the HTML.
2. Take the content from the JavaScript file and place it inside a `<script>` tag at the end of the `<body>` section of the HTML.
3. Remove the original `<link>` tag that pointed to the external CSS file.
4. Remove the original `<script>` tag that pointed to the external JavaScript file.

Here are my files:

`index.html`:

[Paste the full content of your index.html file here]

`css/style.css`:

[Paste the full content of your style.css file here]

`js/script.js`:

[Paste the full content of your script.js file here]

Provide the complete, final `index.html` file as your response."

Step 3: Save and Verify the Final File

The AI will generate a single block of HTML code. This new code should contain all of your original HTML structure, with the contents of your CSS and JavaScript files now embedded directly inside `<style>` and `<script>` tags.

1. **Save the File:** Copy the AI's response and save it as a new file. Give it a versioned name, like `my-app-v1.0.html`, to distinguish it from your development files.
2. **Final Test:** Double-click this new, single file to open it in your browser. Perform a final function check to ensure that everything still works exactly as it did before. The functionality should be identical.

You now have a robust, single-file application that is ready for your documentation packet and for distribution.

Advanced Technique: Inlining Images

For true single-file portability, you can even embed images directly into your file. This is done by converting the image into a special text format called a "data URL." This is an excellent technique for small images like logos or icons.

You can ask your AI to do this for you.

Example Prompt for Inlining an Image:

> "In the provided HTML, find the `<img>` tag that references `logo.png`. Please convert the `logo.png` image file into a data URL and embed it directly into the `src` attribute of the `<img>` tag."

You would then need to provide the AI with the image file (if the interface supports it) or use a separate online tool to convert the image to a data URL and paste it into the prompt. While powerful, this technique can make your HTML file very large if used for big images, so it's best reserved for small visual elements.

Building a SPA in a single HTML file in vanilla JavaScript by Jesse Pence:
http://www.youtube.com/watch?v=tsHFUKKdy8Y

Chapter 16: Versioning & Distribution

Concept: The finished version needs to be clear with a safe distribution path.

You've built your application, tested it, secured it, and compiled it into a single, portable HTML file. The final step is to get it into the hands of the people who need it. Just like any official document or piece of equipment, a software tool needs a standardized process for versioning and distribution. This ensures that everyone is using the correct version, that changes are tracked, and that the tool is shared through approved, secure channels.

This chapter covers the best practices for packaging, versioning, and sharing your application, as well as how you or others can update it in the future.

1. Versioning: Why It's Non-Negotiable

Imagine getting an update to a tactical manual, but there's no version number. You would have no way of knowing if you have the latest information or if you're working off an old, outdated procedure. Software is exactly the same. Versioning is how you track changes, fix bugs, and add features in a clear and organized way.

For our tools, we use a simple and standard system called **Semantic Versioning (SemVer)**. It uses a three-part number: **MAJOR.MINOR.PATCH**.

- **PATCH (e.g., 1.0.1):** Update like this for small bug fixes that don't change any features.
- **MINOR (e.g., 1.1.0):** Update like this when you add a new feature that is backward-compatible (it doesn't break how the old features worked).
- **MAJOR (e.g., 2.0.0):** Update like this for major changes that could break older versions.

How to Apply Versioning:

When you save your final, compiled HTML file, name it with its version. This is the simplest and most effective way to keep track of your releases.

Example: `checklist-app-v1.0.0.html`

When you fix a small bug, your next release will be `checklist-app-v1.0.1.html`. When you add a new "Export to PDF" feature, your next release will be `checklist-app-v1.1.0.html`.

2. Packaging and Sharing Your Application

With your versioned file ready, you need to share it through official, approved channels. Never use personal email or unauthorized file-sharing sites.

How to Package:

Your primary output is the single, compiled `.html` file you created in the previous chapter. This is often all you need. If your project includes supporting documents (like your `docs/` folder or sample data files), it's best to put everything into a single `.zip` folder and name it with the version number (e.g., `my-app-v1.2.0.zip`).

How to Share:

Use command-approved methods for distribution. The best options are typically:

- **A SharePoint or Flank Speed site.**
- A command-approved intranet page or a repository for unclassified tools.
- Official command email, if the file size is acceptable.

By using these channels, you ensure that your tool is distributed securely and that there is an official record of its release.

3. The Reverse Process: "Decompiling" for Future Updates

The single, compiled `.html` file is perfect for sharing, but it's not ideal for making updates. It's a large, complex file where the structure, style, and logic are all mixed together.

So, what happens when a user finds a bug or you want to add a new feature? You go back to your original, clean, **structured folder**. But what if you don't have it anymore, or if you're handing the project off to someone else?

This is where the concept of **decompiling** comes in. The "manifest" that the Warfighter Coder Tool (WCT) embeds in your compiled file acts as a set of instructions. A tool with a "decompiler" can read this manifest and automatically reverse the compilation process, unpacking the single `.html` file back into its original, organized structure:

`my-app-v1.2.0.html` -> **Decompile** -> `my-app/`, `css/`, `js/`

Why This is Powerful:

This capability means that your distributable file is also your source code backup. Anyone with the right tool can take your final product, decompile it back into a clean development environment, make their changes, and then re-compile it into a new version (e.g., `v1.2.1`). This makes your tools sustainable, maintainable, and easy to collaborate on, ensuring they can evolve to meet new needs long after you've built the first version.

This completes the development lifecycle: you develop in a clean, structured folder, compile it for simple distribution, and anyone can decompile it to continue the cycle of improvement.

It's time to fix semantic versioning by Theo - t3.gg: http://www.youtube.com/watch?v=5TIDnT9LTFc

Part 5: Advanced Capabilities & Next Steps

(Focus: Expanding your skills beyond single-user, offline apps and introducing powerful tools.)

Chapter 17: Multi-User Collaboration with ShareDrive-NoSQL

> **Concept: Use a shared JSON file on a network drive as a simple database for team collaboration.**

So far, we have built powerful, self-contained applications that run completely offline. We've made them remember a user's data between sessions using `localStorage`, and we've added a data lifeline with Import and Export functions. This is perfect for a tool used by a single person. But what happens when an entire team needs to see and interact with the same live data?

The `localStorage` "file cabinet" we used in Chapter 11 is private. It belongs to one user on one specific browser on one machine. It's impossible for a teammate on another computer to see the changes you've made. To enable collaboration, we need to graduate from a private file cabinet to a shared, central logbook.

This chapter introduces a powerful "Phase 2" for your applications: turning them into multi-user tools by using a single JSON file on a shared network drive as a simple, live database.

Introducing the "ShareDrive-NoSQL" Concept

This approach allows your static HTML application to read from and write to a central file that everyone on your team can access. Think of it like a digital version of a maintenance logbook, a flight schedule, or a NATOPS binder that sits on a desk in the ready room.

- One person opens the logbook and writes an entry.
- They close the logbook.
- The next person who opens it sees the new entry immediately.

We can achieve this digitally without a traditional server by using a shared network drive (like a SharePoint site or a common file server) and a modern browser feature called the **File System Access API**. This allows a web page, *with the user's explicit permission*, to get a persistent handle to a file and read or write to it directly.

A helper library, often named something like `sharedrive-nosql.js`, can manage this process for you, handling the complexities of file access, polling for changes, and merging concurrent edits.

How It Works: The Read-Merge-Write Cycle

When you refactor your application to use this model, it follows a continuous cycle to keep data in sync.

1. **Opening the "Database":** The user clicks a new button in your app, like "Open Shared Watch Log." Their browser will open a file picker, and they will navigate to the shared network drive and select the central `database.json` file. Your application now has a secure handle to that file.
2. **Polling for Changes:** In the background, your app's JavaScript will periodically check the shared file's "last modified" timestamp. If the timestamp is newer than the last time the app checked, it knows that someone else on the team has saved an update.
3. **Merging:** When a change is detected, the app reads the new content from the shared file and intelligently **merges** it with the data it currently has in memory. This is the critical step for collaboration. It doesn't just blindly overwrite your local data; it compares the two versions and combines them, typically using a "last-write-wins" strategy for any conflicting edits.
4. **Local Changes:** When the user interacts with the app in their own browser (e.g., adds a new checklist item), the change is applied instantly to their local in-memory version of the data.
5. **Saving:** After a local change is made, the application waits for a brief moment (a "debounce" period) and then writes its new, merged version of the data back to the shared JSON file, updating the "last modified" timestamp for everyone else.

This cycle of polling, merging, and saving allows a team to collaborate on a single data source using a simple, serverless architecture.

Workflow: Refactoring Your App with an AI

Transitioning an application from `localStorage` to a shared file model is a significant architectural change. This is a perfect task for your AI pair programmer, as it involves refactoring existing code to work with a new data-handling library.

Step 1: Get the `sharedrive-nosql.js` Library

The first step is to add the helper library to your project. The Warfighter Coder Tool (WCT), which we will discuss later, has a built-in feature to automatically add this file to your `js/` folder.

Step 2: Craft a Refactoring Prompt

You will provide your AI with your existing application code, the code for the `sharedrive-nosql.js` library, and a clear set of instructions to perform the integration.

Example Refactoring Prompt:

> "You are an expert front-end engineer. Your task is to refactor my existing checklist application to use the provided `sharedrive-nosql.js` library for multi-user collaboration instead of `localStorage`.
>
> **Task:**
>
> 1. Add two new buttons to the `index.html` file: "Open Shared Checklist" and "Create New Shared Checklist."
> 2. In `script.js`, remove all code that uses `localStorage.setItem()` and `localStorage.getItem()`.
> 3. Initialize a new `FsNoSqlDB` instance from the `sharedrive-nosql.js` library.
> 4. Connect the new buttons to the `db.open()` and a function to create a new one.
> 5. Modify the existing `saveState()` function to instead use `db.put()` to save records to the shared database.
> 6. Modify the `loadState()` function to read data from `db.snapshot()` instead of `localStorage`.
> 7. Add a listener for the `db.onRemoteMerge` event that re-renders the checklist when new data comes in from another user.
>
> **Here are my files:**
>
> **`index.html`:**

[Paste your full index.html here]

> **`js/script.js` (Current version):**

[Paste your current script.js here]

> **`js/sharedrive-nosql.js` (The library to use):**

[Paste the full content of the library here]

> Provide the completely updated `index.html` and `script.js` files as your response."

This prompt gives the AI a clear, step-by-step set of instructions and all the context it needs to perform a complex refactoring task. By leveraging your AI partner in this way, you can upgrade your personal tool into a true team asset, unlocking a new level of collaborative capability.

WebSockets in 100 Seconds & Beyond with Socket.io by Fireship: http://www.youtube.com/watch?v=1BfCnjr_Vjg

Chapter 18: Adding a Map: Offline GeoJSON with Leaflet

> **Concept: Build offline maps without external tile servers by converting geographic data into a local JavaScript file.**

Many tactical applications rely on maps to visualize information, from tracking assets to planning routes. In a connected world, this is simple: web maps pull down thousands of small image "tiles" from servers to display a map of the world. In our offline environment, this is impossible. We have no connection to a tile server.

The solution is to shift our thinking from image-based maps to data-based maps. Instead of downloading pictures of coastlines, we will draw them ourselves using raw geographic data. This chapter will teach you how to use **GeoJSON**, a standard format for geographic data, and **Leaflet.js**, a popular mapping library, to create fully interactive, offline-capable maps for your applications.

Understanding the Technologies

1. **GeoJSON: The Digital Map Data**
   GeoJSON is a text-based format for representing geographic shapes, built on the JSON structure we learned about in Chapter 5. Think of it as a list of digital coordinates that describe points, lines, and polygons. A GeoJSON file might contain the outline of a country, the path of a planned route, or the location of a specific point of interest. Because it's just text, it's lightweight and easy to work with.
2. **Leaflet.js: The Map Engine**
   Leaflet.js is a free, open-source JavaScript library that does one thing very well: it creates fast, interactive maps. It acts as our map engine, taking our GeoJSON data and rendering it as clickable, zoomable shapes on the screen. It is the most popular mapping library for a reason: it's simple, powerful, and easy to use.

The Offline Mapping Workflow

To create an offline map, we will follow a four-step process. Our goal is to take a standard GeoJSON file from the internet and convert it into a local JavaScript file that our application can use without needing a network connection.

Step 1: Get Your Geographic Data (GeoJSON)

First, you need the raw data for the map you want to display. There are many sources for this, but a great place to start is a public repository that offers downloadable GeoJSON files for countries and regions.

- **Action:** Go to a site like geojson-maps.kyd.au and download a GeoJSON file. For a world map, navigate to *World -> Countries*.

- **Pro Tip:** Choose a "low" or "medium" resolution file. Higher resolution means larger file sizes, which can slow down your application. For most use cases, low resolution is perfectly adequate. Save this file to your computer (e.g., `countries.geojson`).

Step 2: Convert GeoJSON into a JavaScript File

This is the most critical step for offline use. A browser has security restrictions that prevent a local `index.html` file from easily loading another local file like `countries.geojson`. We work around this by turning our data into a JavaScript variable.

We will ask our AI to wrap the GeoJSON content inside a JavaScript file. This "tricks" the browser into loading our map data as if it were part of our application's code.

Example Conversion Prompt:

> "Take the following GeoJSON content and wrap it in a JavaScript file. The file should create a single global variable named `worldGeoData` that holds the GeoJSON object.
>
> **GeoJSON Content:**

[Paste the full content of your downloaded .geojson file here]

> Provide the complete, final `.js` file as your response."

The AI will return a JavaScript file that looks like this:

`world-data.js` (AI-generated output):

```
// Generated by AI for offline map use
window.worldGeoData = {
  "type": "FeatureCollection",
  "features": [
    // ... all of your geographic data will be here ...
  ]
};
```

Save this new file inside your project's `js/` folder (e.g., as `js/world-data.js`).

Step 3: Integrate the Map into Your HTML

Now, you'll modify your `index.html` to include the Leaflet library, your new data file, and a place for the map to live.

Example `index.html` additions:

```
<head>
```

```
    <!-- ... your other tags ... -->

    <!-- 1. Add Leaflet's CSS and JavaScript (from a CDN for now) -->
    <link rel="stylesheet"
href="[https://unpkg.com/leaflet@1.9.4/dist/leaflet.css](https://unpkg.com/leaflet@1.9.4/dist/leafl
et.css)" />
    <script
src="[https://unpkg.com/leaflet@1.9.4/dist/leaflet.js](https://unpkg.com/leaflet@1.9.4/dist/leaflet.j
s)"></script>

    <!-- 2. Add your local GeoJSON data file -->
    <script src="js/world-data.js"></script>

    <!-- 3. Add some basic CSS for your map container -->
    <style>
        #map {
            height: 500px;
            width: 100%;
            background-color: #1a2a3a; /* A dark blue for the "ocean" */
        }
    </style>
</head>
<body>
    <!-- ... your other app content ... -->

    <!-- 4. Add a container for the map to live in -->
    <div id="map"></div>

    <!-- ... your main script.js tag ... -->
</body>
```

Step 4: Prompt Your AI to Render the Map

Finally, you will instruct your AI to write the JavaScript code that initializes Leaflet and draws your data. This is where you specify that you do *not* want an online tile layer.

Example Map Rendering Prompt:

> "Using the Leaflet.js library, write the JavaScript code to render an offline map.
>
> **Task:**
>
> 1. Initialize a new Leaflet map inside the `<div id="map"></div>`.
> 2. **Do not add a tile layer.** The map background should just be the color set in the CSS.
> 3. Load the GeoJSON data from the `window.worldGeoData` variable.
> 4. Style the polygons with a blue fill and a light-colored border.
> 5. When a user hovers over a country, show its name in a tooltip.

6. Automatically zoom and center the map to fit all the loaded data.

Provide the complete JavaScript code for my `script.js` file."

The AI will generate the logic to draw your map. You'll add this to your `js/script.js` file, save it, and when you open your `index.html`, you will see a fully interactive, offline map. This same process can be used to display tactical overlays, routes, or any other geographic information your mission requires.

leaflet js to create your map offline (cross platform) html5 by Wonder Developer:
http://www.youtube.com/watch?v=oP4bCLtXIeY

Part 6: Reference & Next Steps

Chapter 19: The Pattern Library & Troubleshooting

> **Concept: Most tools are variations of a few common patterns. Knowing these patterns and how to fix them is the final step in becoming a self-sufficient builder.**

You have now walked through the entire development lifecycle, from defining a problem to building, securing, and distributing a functional application. As you build more tools, you will notice that most operational problems can be solved with a handful of common application "patterns." A tool to track maintenance is just a variation of a checklist; a tool for calculating fuel burn is a specialized calculator.

This chapter provides a library of these common patterns, complete with starter prompts you can adapt for your own missions. It also includes a simple troubleshooting guide to help you diagnose and fix the most common issues you will encounter, turning moments of frustration into opportunities for learning.

Part 1: The Pattern Library

You don't need to reinvent the wheel for every new problem. Use these patterns and their corresponding prompts as a starting point for your own projects.

1. The Checklist / Tracker

- **Core Idea:** A list of items with a status (e.g., complete, incomplete, N/A). This is the most fundamental pattern for tracking tasks, inspections, and procedures.
- **Tactical Use Cases:** Pre-flight inspections, watch turnover checklists, maintenance procedures, personnel qualification tracking.
- **Starter Prompt:**
  "Create a single-file HTML checklist app for a pre-dive checklist. The app must work completely offline and save its state to `localStorage`. Include an 'Export to CSV' button to save the status of all tasks. The tasks are:
  ."

2. The Calculator / Converter

- **Core Idea:** Takes user input, performs a calculation, and displays the result. This pattern is for any tool that automates a mathematical or logical process.
- **Tactical Use Cases:** Fuel burn calculators, unit conversion tools (e.g., nautical miles to kilometers), time-on-target calculators, weight and balance calculators.
- **Starter Prompt:**
  "Create a single-file HTML fuel consumption calculator for an aircraft. It needs input fields for 'Flight Hours' and 'Fuel Burn Rate (gallons/hour)'. It should calculate the 'Total

Fuel Required' and display it in real-time. The app must work offline and have a 'Clear' button to reset the fields."

3. The Dashboard

- **Core Idea:** Visualizes data from an imported file (like a CSV or JSON). This pattern is for turning raw data into actionable intelligence through charts and tables.
- **Tactical Use Cases:** Visualizing maintenance logs, displaying sortie statistics from an exported CSV, creating a simple dashboard for sensor data.
- **Starter Prompt:**
  "Create a single-file HTML dashboard app that uses the Chart.js library (from a CDN for now). The app must have a file input that accepts a CSV file. When a user uploads a CSV with 'Date' and 'Sorties Flown' columns, it should draw a bar chart showing the number of sorties flown per date."

4. The Form / Worksheet

- **Core Idea:** A guided, multi-step form for collecting structured information from a user. This is perfect for standardizing data entry for reports or logs.
- **Tactical Use Cases:** A digital after-action report, a site survey worksheet, an equipment checkout form.
- **Starter Prompt:**
  "Create a multi-step wizard form for a post-mission report. It should have three steps: 'Mission Details', 'Events', and 'Summary'. Use JavaScript to show and hide the steps with 'Next' and 'Previous' buttons. At the end, show a summary page with all the information the user entered and provide a 'Print to PDF' button. This must be a single, offline HTML file."

Part 2: The Troubleshooting Guide

Every builder, from novice to expert, runs into bugs. Frustration is part of the process. The key is to have a systematic way to diagnose and solve problems. This guide covers the most common issues you'll face.

Your Primary Tool: The F12 Developer Console

As we covered in Chapter 10, your first and most important action when something goes wrong is to **press F12** and check the **Console** for error messages. An error message is not a sign of failure; it is a clue. It is the single most valuable piece of information you can give your AI to help it fix the problem.

Common Problems and How to Fix Them

Problem	Likely Cause	Solution / What to Tell Your AI
Blank White Screen / App Doesn't Load	A "fatal" JavaScript error is preventing the page from rendering. This is often a syntax error or a typo.	1. Press **F12** and look for a red error in the Console. 2. Prompt your AI: *"My app is showing a blank screen. Here is the error from the console and my full code. Please fix it."*
Styling is Missing or Looks Wrong	The CSS rules are not being applied correctly. This could be a typo in a CSS selector (e.g., `.my-buton` instead of `.my-button`) or a problem with how the `<style>` tag is set up.	1. Check your CSS selectors against the `id` and `class` attributes in your HTML. 2. Prompt your AI: *"The styling in my app is broken. Here is my full code. Please check the CSS selectors and fix any issues."*
Buttons or Controls Don't Do Anything	The JavaScript event listener is not attached correctly. This usually means the `id` in your JavaScript (`getElementById("my-btn")`) doesn't match the `id` in your HTML (`<button id="my-button">`).	1. Compare the `id` used in your `addEventListener` code with the `id` in your HTML button tag. 2. Prompt your AI: *"My button with the id 'my-btn' is not working when I click it. Here is my HTML and JavaScript. Please find and fix the event listener."*
Data Doesn't Save After Refreshing	There's an issue with your `localStorage` code. You might be saving the data correctly but failing to load it, or vice versa.	1. Press **F12**, go to the "Application" tab, and look under "Local Storage" to see if your data is actually being saved. 2. Prompt your AI: *"My app is not remembering its state after I refresh the page. Here is my full code. Please review my*

		`localStorage` *save and load functions and fix them."*
Import/Export Buttons Don't Work	The code for creating or reading the file has a bug. This can also be caused by browser security settings, though modern browsers are quite permissive.	1. Check the **F12 Console** for errors when you click the buttons. 2. Prompt your AI: *"My 'Export to CSV' button is not downloading a file. Here is my full code and the error message from the console. Please fix the export function."*

Remember the core workflow: **Test -> Triage (F12) -> Refine**. By following this simple loop and providing your AI with clear, specific information, you can solve almost any problem you encounter.

10 Design Patterns Explained in 10 Minutes by Fireship: http://www.youtube.com/watch?v=tv-_1er1mWI

Chapter 20: Your Force-Multiplier: The Warfighter Coder Tool (WCT)

> **Concept: Now that you know the manual process, the WCT is an all-in-one IDE that automates and simplifies every workflow taught in this handbook.**

Throughout this handbook, you have learned the fundamental, tool-agnostic methods for building software. You have learned how to define a problem, structure a project, write precise prompts for an AI, and iterate on the code using nothing more than a text editor and a web browser. That manual process is the core skill of a warfighter-coder.

Now, it's time to introduce your force-multiplier: the **Warfighter Coder Tool (WCT)**.

The WCT is a self-contained, browser-based Integrated Development Environment (IDE) designed specifically for the workflows taught in this guide. It is a single HTML file that you can run locally, just like the apps you've been building. It takes every manual step we've covered—creating folders, prompting the AI, testing, compiling, and securing—and provides a dedicated tool to make it faster, easier, and more reliable.

By learning the manual process first, you understand *why* each step is important. Now, with the WCT, you can execute those steps with maximum efficiency.

Key Features of the Warfighter Coder Tool

The WCT is organized around the same lifecycle we have followed: Plan, Build, Harden, and Ship.

1. Plan: The LLM Prompt Tool

- **What it Replaces:** Manually writing out your TAG, JTBD, and TCF prompts.
- **How it Works:** The "Plan" section of the WCT contains an **LLM Prompt Tool**. This is a guided workflow that walks you through the entire problem-definition process. You input your user interview notes, and it helps you generate the Jobs-to-be-Done. You select a job and a tool idea, and it helps you craft the MVP ladder. Finally, it combines all this context into a perfectly formatted TCF build prompt that you can copy with a single click. It automates the strategic thinking process, ensuring you always start with a high-quality prompt.

2. Build: Code Editor, Live Preview & Dev Console

- **What it Replaces:** A separate text editor, manually saving files, and constantly refreshing your browser.
- **How it Works:** The WCT includes a complete, multi-file **Code Editor**. You can create folders and files, write code, and manage your entire project structure from one place.

    - **Live Preview:** As you make changes to your HTML, CSS, or JavaScript, a live preview pane updates instantly. This replaces the slow, manual process of saving your file, switching to your browser, and hitting refresh. It dramatically accelerates the "Test" phase of your build loop.
    - **Dev Console:** The WCT has its own built-in developer console that mirrors the functionality of your browser's F12 tool. Errors and `console.log` messages appear directly within the tool, keeping your entire build loop—Test, Triage, and Refine—inside a single interface.

3. Harden: SAST Scanner & Security Reviewer

- **What it Replaces:** Manually auditing your code for security issues and writing documentation from scratch.
- **How it Works:** The "Harden" section provides tools to automate the security and approval process.
    - **SAST Scanner:** The Static Application Security Testing (SAST) scanner is a one-click tool that reads your JavaScript files and automatically flags common security vulnerabilities, like the use of `innerHTML` or `eval()`, before you even begin a formal audit.
    - **Security Reviewer:** This tool automates the creation of your documentation packet. It helps you generate the ASD STIG prompt from Chapter 14, providing your code and the specific controls to the AI. You can then paste the AI's JSON response back into the tool, which formats it into a clean, readable HTML report ready for your IA team.

4. Ship: HTML Compiler & Decompiler

- **What it Replaces:** The tedious manual process of copying and pasting your CSS and JavaScript into your `index.html` file for distribution.
- **How it Works:** The "Ship" section automates the final packaging of your application.
    - **HTML Compiler:** With one click, the compiler reads your entire structured folder, inlines all your CSS and JavaScript, converts images to data URLs, and bundles everything into a single, portable `.html` file, just as we discussed in Chapter 15.
    - **Decompiler:** The compiler embeds a "manifest" inside the final HTML file. The Decompiler can read this manifest and automatically unpack a single-file app back into its original, structured folder format. This is critical for long-term maintenance, allowing you or another warfighter to easily make changes to the project in the future.

5. Advanced Modules: One-Click Integrations

- **What it Replaces:** The complex, multi-step prompting required to add advanced features like databases and maps.
- **How it Works:** The WCT includes modules for advanced capabilities like **ShareDrive-NoSQL** and **Leaflet Maps**. Instead of manually prompting the AI for all the

boilerplate code, these tools provide a simple interface to automatically add the necessary libraries and starter code to your project. This allows you to integrate complex features in minutes instead of hours.

By understanding the principles in this handbook, you have gained the core knowledge of a developer. With the Warfighter Coder Tool, you now have the professional-grade equipment to apply that knowledge at speed and scale.

Chapter 21: Your 30-Day Mission

Challenge: A clear, 4-step plan to build, test, refine, and share a digital tool.

You have reached the end of this handbook, but you are at the beginning of your journey as a warfighter-coder. You have learned the theory, the tactics, and the procedures for building software. You understand the languages, the security principles, and the development workflow. But knowledge is only potential power. True capability comes from application.

This chapter provides a simple plan to guide you from identifying a real-world problem to deploying your first operational tool. This is your opportunity to create something that makes a real impact—to save someone time, to reduce errors, or to provide clarity. .

Remember the core principle that has guided us from the beginning: **a working, ugly tool is better than a perfect, imaginary one.** Do not wait for the perfect idea or the perfect design. Start now, build something small, and get it into the hands of someone who can use it.

Your 30-Day Mission Plan

This plan breaks down the development process into four manageable, one-week sprints. Each week focuses on a specific phase of the lifecycle you've learned in this handbook.

Week 1: Find the Mission (Discover & Plan)

- **Objective:** Identify a real, tactical problem and create a precise build prompt for your MVP.
- **Actions:**
    1. **Find a User:** Identify one person in your unit—a peer, a junior sailor, a senior leader—who has a small, frustrating problem in their daily workflow. It could be a tedious calculation, a paper-based log, or a confusing checklist.
    2. **Conduct a JTBD Interview:** Sit down with them for 15 minutes. Use the techniques from Chapter 7. Ask "Tell me about the last time you..." and listen for the pain points. Do not pitch a solution. Just listen.
    3. **Define the Problem:** Write down your TAG statement (Target, Actual, Goal) based on your conversation.
    4. **Craft Your Prompt:** Use the TCF pattern (Task, Context, Format) from Chapter 8 to write a clear, concise build prompt for the simplest possible version of a tool that could solve their problem.
- **End State for Week 1:** A single, well-defined TCF prompt saved in a text file.

Week 2: Build the MVP (Build & Test)

- **Objective:** Generate the first working version of your application. The goal is functionality, not beauty.
- **Actions:**

    1. **Generate Code:** Paste your TCF prompt into your AI and generate the code for your `index.html` file (or your structured folder, if you chose that route).
    2. **Save and Run:** Save the code into your project file(s) and open the `index.html` file in your browser.
    3. **The Build Loop:** Use the **Test -> Triage (F12) -> Refine** loop from Chapter 10. Your app will be broken. Use the F12 Console to find the red error messages. Feed the errors and your code back to the AI until the tool is functional.
- **End State for Week 2:** A working, if ugly, `index.html` file that successfully performs its core function without errors.

Week 3: Refine with Feedback (Iterate & Harden)

- **Objective:** Show your MVP to your user, get feedback, and make it more robust.
- **Actions:**
    1. **The Silent Observer Test:** Put your MVP in front of your user. Give them a task and say nothing. Watch where they struggle. Their confusion is your to-do list.
    2. **Iterate:** Take their feedback and use the Build Loop to make one or two key improvements.
    3. **Add Persistence:** Ask your AI to add `localStorage` (Chapter 11) to your app so it can remember the user's data between sessions.
    4. **Prepare for Approval:** Start your documentation. Use your AI to help you draft the `README.md` and `SECURITY-NOTE.md` files from Chapter 14.
- **End State for Week 3:** An improved application that has been validated by a real user and has basic documentation.

Week 4: Ship It (Package & Deploy)

- **Objective:** Package your application for distribution and get it into the hands of your user(s).
- **Actions:**
    1. **Compile Your App:** If you used a structured folder, use the WCT's Compiler (or the manual process from Chapter 15) to bundle your project into a single `index.html` file.
    2. **Version Your File:** Rename your final file with a clear version number (e.g., `pre-flight-checklist-v1.0.0.html`) as described in Chapter 16.
    3. **Final Test:** Do one last run-through of the application to ensure everything works as expected.
    4. **Distribute:** Share the file with your user(s) through SharePoint, email, etc.
- **End State for Week 4:** Your first tool is officially deployed and solving a real problem.

This 30-day mission is your first real contribution as a warfighter-coder. The outcome, a tangible tool that helps your team, is the ultimate reward. Welcome to the cadre.

How to Create a Company | Elon Musk's 5 Rules by Savanteum: http://www.youtube.com/watch?v=Qa_4c9zrxf0

Appendix A: Master Prompt Library

This appendix contains a collection of high-quality, copy-and-paste prompt templates to use as a starting point for your conversations with an AI.

1. Core Build Prompts

Use these prompts to generate the initial structure of your application.

Master TCF Prompt (Single-File App)

This is your go-to prompt for creating simple, self-contained applications. It's perfect for MVPs and small tools.

TASK: Create a single-file HTML application that functions as a [your app's purpose, e.g., checklist tracker].

CONTEXT: This app will be used offline by [describe the user, e.g., a pilot] on a locked-down government laptop with no internet. It must run from a single local HTML file and must not make any external network calls. The user interface should be simple and intuitive.

FORMAT: Provide all the code in a single HTML file. All CSS and JavaScript must be included internally within `<style>` and `<script>` tags. Do not use any external CDN links for libraries or fonts.

Multi-File Prompt: Step 1 (index.html)

Use this when starting a more complex project with a structured folder.

TASK: Create the main HTML structure for a [your app's purpose, e.g., multi-mission tracking application]. It needs a header, a main content area, and a footer.

CONTEXT: This is the main HTML file for an offline application.

FORMAT: Provide only the HTML code for the `index.html` file. It must link to an external stylesheet at `css/style.css` and an external JavaScript file at `js/script.js`. Do not include any inline CSS or JavaScript.

Multi-File Prompt: Step 2 (style.css)

After generating the index.html, use this prompt to create the styles.

TASK: Now, create the CSS for the application based on the HTML structure I provided. Give it a clean, professional dark-theme design suitable for a tactical environment.

CONTEXT: This CSS will style the `index.html` file. The main container has a class of `main-container`.

FORMAT: Provide only the CSS code for the `style.css` file.

Multi-File Prompt: Step 3 (script.js)

Finally, use this prompt to generate the initial JavaScript logic.

TASK: Now, write the JavaScript logic for the application. For now, just make the button with the id `add-mission-btn` log a message to the console when clicked.

CONTEXT: This JavaScript is for the `index.html` file I provided.

FORMAT: Provide only the JavaScript code for the `script.js` file.

2. App-Specific Starter Prompts

Adapt these templates for common application patterns.

Checklist App:

TASK: Create a single-file HTML checklist app with 10 items.

CONTEXT: Offline use, local file.

FORMAT: All code in one HTML file, saving progress to localStorage, with a CSV export button.

Calculator:

TASK: Create a single-file HTML calculator for [describe purpose, e.g., calculating travel time].

CONTEXT: Offline use, local file. Needs input fields for [e.g., distance and speed].

FORMAT: All code in one HTML file. Should show the formula used and save a history of recent calculations to localStorage.

Form / Worksheet:

TASK: Create a single-file HTML form with fields for [list the fields, e.g., name, date, notes].

CONTEXT: Offline use, local file. Data needs to be collected and then exported.

FORMAT: All code in one HTML file. Include data validation to ensure fields are not empty. Provide a button to export the form data as a single JSON object.

Dashboard:

TASK: Create a single-file HTML dashboard to visualize data from a CSV file.

CONTEXT: Offline use, local file. The user will upload a file to view it.

FORMAT: Use the Chart.js library (from a CDN for development). The app must have a file input that accepts a CSV. When a user uploads a CSV with "[Column A]" and "[Column B]" columns, it should draw a bar chart.

3. Feature & Refinement Prompts

Use these prompts to fix bugs, add features, or generate documentation for an existing application.

Fixing an Error:

The most important refinement prompt. Be as specific as possible.

TASK: My application is not working correctly. Please fix it.

CONTEXT: When I click the "Mark Complete" button, nothing happens. Here is the error message from the F12 Developer Console.

FORMAT: Provide a complete, corrected version of the code. Explain what the error was and how you fixed it.

[Paste the exact error message here]

[Paste your full HTML/JS/CSS code here]

Adding localStorage (Memory):

TASK: Update the following application to save its state.

CONTEXT: Here is the full code for my working application.

FORMAT: Modify the JavaScript to use `localStorage` to save all user inputs. When the page is reloaded, the previous state should be restored.

Adding Import/Export:

TASK: Add data import and export functionality to my application.

CONTEXT: Here is my working application code. The core data is stored in a JavaScript variable named `tasks`.

FORMAT: Add two buttons:

1. "Export to JSON": This should download the `tasks` array as a JSON file.

2. "Import from JSON": This should allow a user to select a JSON file, which then replaces the current `tasks` and updates the display.

Generating a README.md:

TASK: Write a `README.md` file for my application.

CONTEXT: Here is the full code for the application. It is a [describe app type, e.g., pre-flight checklist tool]. The user opens the `index.html` file and can [describe core actions].

FORMAT: Generate a clear and concise README file in Markdown format. It should include a brief description, instructions on how to use the app, and a list of its main features.

Appendix B: All Checklists

Use these checklists at each stage of your project to ensure you haven't missed a critical step. They are designed to be a quick reference to keep you on track.

1. Pre-Build Checklist

Complete this before you write your first line of code or your first prompt. A solid plan is the foundation of a successful project.

- [ ] **Define the Mission:** Have I written a clear, one-paragraph problem statement using the **TAG** framework (Target, Actual, Goal)?
- [ ] **Identify the User:** Do I know exactly who this tool is for and what their operating environment is like?
- [ ] **List Constraints:** Have I listed all key constraints (e.g., must work offline, no installation, simple UI)?
- [ ] **Define Success:** Do I know what a successful outcome looks like (e.g., time saved, errors reduced, better decision-making)?
- [ ] **Scope the MVP:** Have I scoped the project down to the smallest possible **Minimum Viable Product (MVP)** that solves the core problem?

2. Security Checklist

Review this checklist every time you interact with a public AI and before you distribute your application. OPSEC is non-negotiable.

- [ ] **Prompt Hygiene:** Have I removed all classified, CUI, PII, or other sensitive information from my prompts?
- [ ] **Data Sanitization:** Have I created sanitized, generic data examples for the AI instead of using real operational data?
- [ ] **No Hardcoded Secrets:** Does my application's final code contain any embedded secrets, passwords, API keys, or sensitive configuration data?
- [ ] **Safe Input Handling:** Does my application safely handle any user-provided data (like from an imported file) to prevent Cross-Site Scripting (XSS)? Use `.textContent` instead of `.innerHTML`.
- [ ] **Fully Offline:** Does the final, compiled code contain any links to external websites (`http://` or `https://`)? (Other than approved CDNs for development).

3. Distribution Checklist

Follow these steps when you are ready to package and share your application.

- [ ] **Package the App:** Is the final application bundled into a single `.html` file?
- [ ] **Version the File:** Is the file named clearly with a version number (e.g., `my-app-v1.2.0.html`)?

- [ ] **Create ZIP Archive:** Have I created a `.zip` file that contains both the final `.html` application and the complete documentation packet (README, SECURITY-NOTE, etc.)?
- [ ] **Final Test:** Have I tested the final, single-file application on a clean machine (or a different browser profile) to ensure it works as expected?
- [ ] **Share Securely:** Am I sharing the `.zip` file through a command-approved channel (e.g., SharePoint, approved email)?

4. Core Functionality Testing Checklist

Use this as a template for your `VERIFICATION.md` document. It covers the essential user actions for a typical offline application.

Test Step	Action	Expected Result
1. Initial Load	Double-click the `index.html` file.	The application loads completely in under 3 seconds without errors.
2. Core Function	Perform the app's main task (e.g., check boxes, enter data).	The user interface updates correctly and shows the expected output.
3. Data Persistence	Enter some data, then refresh the browser page (F5).	The data entered is still present after the page reloads.
4. Data Export	Click the "Export" or "Save" button.	A correctly named file (`.json` or `.csv`) downloads to the user's machine.
5. Clear/Reset	Click the "Clear" or "Reset" button.	All data is cleared from the UI and from `localStorage`.
6. Data Import	Click the "Import" or "Load" button and select the previously exported file.	The application's state is correctly restored to its saved version.

Appendix C: Example Approval Packet

Concept: Make it easy for your local cyber/IA team to say "yes."

To get your application approved for use, you need to anticipate the questions your Information Assurance (IA) or cybersecurity team will ask. By preparing a simple, clear documentation packet, you provide them with all the answers upfront, demonstrating due diligence and making their job easier. This significantly accelerates the approval process.

This appendix provides a complete example packet for a fictional "Pre-flight Checklist App v1.2.0." You should use these documents as templates for your own projects. Create a `docs/` folder inside your project's main folder and place your versions of these files inside it.

1. `README.md` (The "What & How")

The `README` is the front page of your project. It explains what the tool does, who it's for, and how to use it.

```
# Pre-flight Checklist App v1.2.0

**Point of Contact:** [Your Name/Rank/Code]
**Date:** [Date]

## Description

This is a simple, offline checklist application designed to help pilots and aircrew complete and track pre-flight inspections. It allows users to check off items, save their progress locally, and export the final checklist status to a CSV file for record-keeping.

## How to Use

1.  **Package:** The entire application is contained within the `pre-flight-checklist-v1.2.0.html` file.
2.  **Run:** Double-click the `.html` file. It will open in any modern web browser. No internet connection is required.
3.  **Features:**
    * Click checkboxes to mark items as complete.
    * Progress is saved automatically in your browser.
    * Click "Export to CSV" to download the current checklist status.
    * Click "Import from JSON" to load a previously saved state.
    * Click "Reset All" to clear the checklist.
```

2. `SECURITY-NOTE.md` (The Security Statement)

This is the most important document for your IA team. It explicitly states the security posture of your application.

SECURITY NOTE: Pre-flight Checklist App v1.2.0

Application Architecture

This is a purely static HTML/CSS/JS application designed for **complete offline use**.

* **No Server Communication:** The application makes **no external network calls**. It does not "phone home," transmit telemetry, or connect to any external servers or APIs. All code required to run the application is contained within the single `.html` file.
* **Local Execution:** The application runs entirely within the client's web browser using the `file:///` protocol.

Data Handling

* **Data Storage:** All application data (the status of checklist items) is stored locally in the user's browser via `localStorage`. This data is sandboxed by the browser and is not accessible to other websites.
* **Data Persistence:** Data is saved on the user's machine and persists between sessions.
* **Data Deletion:** The user can clear all locally stored data at any time by clicking the "Reset All" button.
* **Input Handling:** The application only handles user input via clicks on pre-defined buttons and checkboxes. When importing a JSON file, the application parses it as data and does not evaluate any content as code, mitigating the risk of Cross-Site Scripting (XSS).

Dependencies

This application has **zero external dependencies**. All necessary HTML, CSS, and JavaScript code is embedded within the single distribution file.

3. `SOFTWARE-BOM.txt` (Bill of Materials)

The BOM lists all third-party software used in your project. For many simple tools, this will be empty. If you use a library (like Chart.js for a dashboard), you must list it here.

Software Bill of Materials (BOM) for Pre-flight Checklist App v1.2.0

This application is built with vanilla HTML, CSS, and JavaScript and contains NO third-party libraries or dependencies.

Example (if you were using a library):
Software Bill of Materials (BOM) for Mission Dashboard v1.0.0

- **Name:** Chart.js
  - **Version:** 3.9.1
  - **License:** MIT License
  - **Source:** Included locally in the compiled HTML file. Originally from https://cdn.jsdelivr.net/npm/chart.js

4. `VERIFICATION.md` (The Test Plan)

This document proves that you have tested the tool's core functionality. It provides a simple, manual test script that anyone can follow to verify that the app works as expected.

Verification & Test Plan for Pre-flight Checklist App v1.2.0

This manual test script verifies the core functionality of the application.

Test Step	Action	Expected Result	Pass/Fail
1. Initial Load	Double-click the `index.html` file.	The app loads completely in under 3 seconds with all checklist items visible.	
2. Core Function	Click the checkboxes for three different items.	The items are visually marked as checked.	
3. Data Persistence	Refresh the browser page (F5).	The three previously checked items are still checked.	
4. Data Export	Click the "Export to CSV" button.	A `.csv` file named `checklist_status.csv` downloads successfully.	
5. Data Reset	Click the "Reset All" button.	All items are cleared and appear unchecked. Refreshing the page confirms they are still unchecked.	

5. `CHANGELOG.md` (The Version History)

A changelog is a simple log of what has changed between versions. This is crucial for maintenance and for letting users know what's new.

Changelog

v1.2.0 - [Date]

* **Feature:** Added "Import from JSON" button to allow users to restore a previously saved state.
* **Fix:** Improved styling on the "Export" button for better visibility.

v1.1.0 - [Date]

* **Feature:** Added "Export to CSV" functionality.
* **Change:** Renamed the app from "Checklist Tool" to "Pre-flight Checklist App".

v1.0.0 - [Date]

* Initial release.
* Core functionality: checklist with `localStorage` persistence.