

Building Modern Web Applications in C

From Sockets to Production: A Complete Guide
to Designing, Implementing, Securing and Operating
Web Software with C



Sockets



Security



Database



Deployment



Monitoring

CALEB ANDERSON

Building Modern Web Applications in C

From Sockets to Production: A Complete Guide to
Designing, Implementing, Securing and Operating Web
Services with C

Steve Publications

This book is available at

<https://leanpub.com/buildingmodernwebapplicationsinc>

This version was published on 2026-10-15



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Steve Publications

Contents

From Sockets to Production: A Complete Guide to Designing, Implementing, Securing and Operating Web Services with C	1
Introduction: Why C for the Web?	2
The State of C in Web Development Today	2
Where C Excels and Where It Does Not	3
How This Book Is Organized	4
The Example Application Journey	5
Development Environment and Conventions	6
Chapter 1: The Web, HTTP and How Requests Flow	8
How a Browser Request Reaches Your Server	8
HTTP/1.1 Message Structure	9
Methods, Headers, Status Codes and Semantics	10
Connection Management and Keep-Alive	11
HTTP/2 and HTTP/3 at a Glance	12
Summary	13
Chapter 2: C, Build Systems and Project Layout	14
Compilers, Standards and Portability Choices	14
Build Systems: Make, CMake and When to Use Each	15
Project Directory Structure and Modular Design	16
Coding Conventions and Style for C Web Projects	18
Installing Dependencies and Managing Libraries	19
A Minimal CMake Configuration	20
Summary	23
Chapter 3: Sockets and Network Programming in C	24
Creating and Configuring a TCP Server Socket	24
Accepting Connections and Handling Clients	24
Blocking, Non-Blocking and Interrupted Syscalls	24

CONTENTS

Reading and Writing: Buffer Handling and Partial Operations	24
Graceful Shutdown and Connection Cleanup	24
Summary	24
Chapter 4: Parsing HTTP Requests	26
Designing a Request Parser	26
Reading Until CRLF: Line-Based Parsing	26
Parsing Headers into a Structured Map	26
Handling the Message Body and Content-Length	26
Error Handling and Malformed Requests	26
Summary	26
Chapter 5: Building a Minimal Web Server	28
Server Initialization and the Main Loop	28
Request Handling and Response Generation	28
Serving Static Content from Disk	28
Returning JSON Responses	28
Testing the Server with curl	28
Summary	28
Chapter 6: Routing and URL Dispatch	30
Route Registration and Handler Signatures	30
Exact Matches, Parameter Extraction and Wildcards	30
Trie-Based Route Matching for Performance	30
404 Handling and Method Not Allowed	30
Organizing Handlers Across Modules	30
Summary	30
Chapter 7: Middleware and Request Processing Pipelines	32
The Middleware Pattern and Execution Flow	32
Writing Logging and Timing Middleware	32
CORS Handling	32
Error Propagation and Short-Circuiting	32
Composing Middleware Chains	32
Summary	32
Chapter 8: Templating and Dynamic HTML	34
Server-Side Rendering in C	34
Using Mustache or a Similar Template System	34
Template Loading and Caching	34

CONTENTS

Escaping and Preventing Injection	34
Generating Complex Pages from Data	34
Summary	34
Chapter 9: JSON Processing	36
Choosing a JSON Library: cJSON vs Jansson	36
Parsing JSON Request Bodies	36
Building JSON Responses Programmatically	36
Error Handling and Validation	36
JSON Schema and Request Validation	36
Summary	36
Chapter 10: Database Integration with SQLite	38
Why SQLite for C Web Applications	38
Opening Databases and Connection Lifetimes	38
Prepared Statements and Parameter Binding	38
Transactions and Concurrency	38
Schema Design and Migration Strategies	38
Summary	38
Chapter 11: Building RESTful APIs	40
REST Principles and Resource Design	40
CRUD Operations and HTTP Semantics	40
Pagination, Filtering and Sorting	40
API Versioning Strategies	40
Error Responses and Consistent Envelopes	40
Summary	40
Chapter 12: Authentication and Authorization	42
Authentication vs Authorization	42
Password Hashing with Argon2 or bcrypt	42
Session-Based Authentication	42
JWT and Stateless Tokens	42
Role-Based Access Control Implementation	42
Summary	42
Chapter 13: Concurrency and Asynchronous I/O	44
Threading Models: Thread-Per-Request vs Thread Pools	44
Implementing a Worker Thread Pool	44
Event Loops and Non-Blocking I/O with epoll/kqueue	44

CONTENTS

Using libuv for Cross-Platform Evented I/O	44
Choosing the Right Concurrency Strategy	44
Summary	44
Chapter 14: Memory Management and Resource Safety	46
Ownership Semantics and Transfer Patterns	46
Arena Allocation for Request-Scoped Data	46
Reference Counting and Shared Resources	46
Detecting Leaks with Valgrind and ASan	46
Defensive Coding Against Buffer Overflows	46
Summary	46
Chapter 15: Error Handling, Logging and Observability	48
Designing Error Codes and Error Propagation	48
Centralized Error Handling in the Request Pipeline	48
Structured Logging with JSON Output	48
Log Levels, Correlation IDs and Debugging	48
Integration with Log Aggregation Tools	48
Summary	48
Chapter 16: Configuration, File Uploads and Caching	50
Loading Configuration from Files and Environment	50
Parsing and Validating Configuration	50
Handling Multipart File Uploads	50
In-Memory Caching Patterns	50
Response Caching and Cache-Control Headers	50
Summary	50
Chapter 17: Secure Coding and Web Vulnerabilities	52
Input Validation and Sanitization	52
Cross-Site Scripting (XSS) Prevention	52
SQL Injection Defense	52
Buffer Overflow and Use-After-Free Mitigations	52
Secure Cookie Handling and Headers	52
Summary	52
Chapter 18: TLS, HTTPS and OpenSSL	54
Why TLS Is Non-Negotiable	54
Setting Up Certificates and Private Keys	54
TLS Handshake and Cipher Suites	54

CONTENTS

Integrating OpenSSL with the Server	54
HTTP to HTTPS Redirection	54
Summary	54
Chapter 19: Testing and Debugging C Web Applications	56
Unit Testing Frameworks: CMocka and Unity	56
Writing Tests for Handlers and Routes	56
Integration Testing Against a Running Server	56
Property-Based Testing and Fuzzing	56
Debugging Techniques and Tools	56
Summary	56
Chapter 20: Performance Optimization and Profiling	58
Measuring Performance: Benchmarks and Metrics	58
Profiling with perf, gprof and Flame Graphs	58
Reducing Allocation and Copying	58
Efficient I/O and Buffering	58
Latency vs Throughput Trade-offs	58
Summary	58
Chapter 21: Deployment Architecture	60
Reverse Proxies: Nginx and Caddy	60
Process Management with systemd	60
Static vs Dynamic Linking for Deployment	60
Reverse Proxy Configuration for Your Server	60
Load Balancing and Multi-Instance Deployments	60
Summary	60
Chapter 22: Containerization and CI/CD	62
Dockerfile Best Practices for C Applications	62
Multi-Stage Builds for Smaller Images	62
GitHub Actions and CI Pipelines	62
Automated Testing in CI	62
Deployment Strategies and Rollbacks	62
Summary	62
Chapter 23: Monitoring and Production Maintenance	64
Health Checks and Readiness Probes	64
Metrics Export and Prometheus Integration	64
Structured Logging at Scale	64

Handling Incidents and Debugging in Production	64
Updates, Hotfixes and Rolling Deployments	64
Summary	64
Conclusion: Beyond the Book	66
The Journey So Far: From Socket to Production	66
Advanced Topics Worth Exploring	66
The Future of C and Web Development	66
Continuing Your Practice	66
Appendix A: Glossary of Essential Terms	67
Appendix B: Quick Reference	68
Essential Build Commands	68
Common HTTP Status Codes	68
Project Structure	68
Security Checklist	68
References	69
HTTP and Web Standards	69
Libraries and Tools	69
Build Systems	69
Security	69
Nginx and Deployment	69
GitHub Actions and CI/CD	69
Prometheus and Monitoring	70
Performance and Benchmarks	70
Additional Resources	70

From Sockets to Production: A Complete Guide to Designing, Implementing, Securing and Operating Web Services with C

About this book

This book teaches you to build, secure, deploy and operate real web applications in C, starting from raw POSIX sockets and growing, layer by layer, into a production system with a database, sessions, a JSON API, TLS, containers and monitoring. Along the way it explains the HTTP standards, the algorithms, the data structures and the security mechanics that make the web work and it shows why C's discipline around memory, buffers and ownership is the same discipline that the web's core infrastructure is built on. The reader is expected to know basic C and a POSIX shell; everything else, from `socket()` to `systemd` to `Argon2id`, is built up from first principles with complete, compilable examples. The book culminates in the deployment of Taskforge, a small team task tracker that evolves through every chapter, from a forty-line server to a hardened, observable, containerized service.

Introduction: Why C for the Web?

There was a time when C dominated the backend. Apache HTTP Server was written in C. Early versions of Nginx were C projects. Databases like MySQL, PostgreSQL and SQLite were built in C. The language that powers operating systems and embedded devices was also the lingua franca of the web infrastructure layer.

Then something shifted. Higher-level languages with dynamic typing, massive package ecosystems and developer-friendly abstractions took over application development. JavaScript with Node.js brought full-stack consistency. Python and Ruby offered rapid prototyping and expressive APIs. Go emerged as a systems language that felt more approachable than C while preserving performance characteristics. Java matured into the backbone of enterprise backend development.

If C was losing ground in web application development, then why is anyone still choosing it today? And more importantly, why should you?

The State of C in Web Development Today

The short answer is that C has never really left the web. It simply moved down a layer. The vast majority of the world's traffic still flows through C-written infrastructure: Nginx, HAProxy, OpenSSL, Redis and SQLite all run in the background of web architectures regardless of what language your application layer is written in.

At the application layer, C has found a different kind of relevance. Performance-sensitive services, latency-critical systems and resource-constrained environments continue to turn to C not because it is fashionable, but because the mathematics of memory, CPU cycles and allocation overhead do not lie. When every microsecond matters, when memory is genuinely limited, or when you need deterministic behavior without garbage collection pauses, C delivers.

Modern C web development is also not what it was twenty years ago. Build systems like CMake have matured into sophisticated, portable tools. Package

managers and dependency tools make library integration easier. Debugging tools such as AddressSanitizer, Valgrind and modern profilers catch bugs that previously took days to track down. Testing frameworks exist, documentation is accessible and the ecosystem of libraries covers everything from HTTP parsing to JSON processing to cryptographic operations.

C is not the tool for every web project. This book will be honest about where C makes sense and where it does not. But for the projects where it does make sense, modern tooling makes it more practical and maintainable than ever before.

Where C Excels and Where It Does Not

Understanding when to use C for web development requires honesty about its strengths and weaknesses. The language is not neutral in its trade-offs. It demands certain things from you and it gives certain things back.

C excels at low-latency request handling because it has no garbage collector, no virtual machine overhead and no runtime surprises. You control exactly where memory goes, exactly when it is freed and exactly what CPU instructions your code produces. When performance is measured in operations per second and every millisecond of latency is a lost user, C consistently ranks among the fastest choices available. Framework benchmarks from TechEmpower regularly show C and C++ frameworks achieving the highest throughput and lowest latency across various benchmarks.

C also excels at memory efficiency. Web applications written in C use a fraction of the memory required by equivalent applications in languages with virtual machines or garbage collectors. This matters for containerized deployments where you want more instances per machine, for edge deployments where hardware is constrained and for any environment where resource costs are a genuine concern.

Another advantage is determinism. C programs behave predictably. There are no hidden background threads doing garbage collection, no JIT warmup phases and no runtime surprises. When you profile a C web application, you see exactly what is happening and you can reason about performance characteristics with mathematical precision.

C is also ideal for building protocol-level services, custom HTTP servers and infrastructure components where understanding and controlling the wire

format matters. When you need to implement a protocol extension, optimize packet processing, or understand the interaction between your application and the network stack at a deep level, C puts you close enough to the metal that nothing feels abstract or invisible.

Where C does not excel is rapid development for common web application patterns. Building a CRUD application with user authentication, file uploads and HTML forms is straightforward in higher-level frameworks and painfully manual in C. The language provides no built-in templating engine, no built-in ORM, no built-in dependency injection and no built-in session management. You either implement these yourself or integrate libraries that may not match the polish or convenience of higher-level framework offerings.

C is also unforgiving of mistakes. Buffer overflows, use-after-free errors, memory leaks and race conditions are all possible bugs in C that are either impossible or much harder to introduce in managed languages. These are not theoretical concerns. Memory safety vulnerabilities account for a significant portion of security issues in C code. Writing safe C requires discipline, good tooling and attention to detail that some teams may not have the resources to provide.

The honest assessment is that C is an excellent choice when performance, memory efficiency, or low-level control is a primary requirement and when the team has (or is willing to develop) the discipline to write safe systems code. It is a poor choice for teams that prioritize rapid iteration, developer convenience, or when the performance advantages do not justify the additional engineering complexity.

How This Book Is Organized

This book takes a progressive, practical approach. We start with fundamentals and build upward, always writing real code rather than discussing abstract concepts. Every chapter adds concrete capability to your knowledge and to the example applications we develop together.

The material is organized in roughly chronological order of how you would build a real C web application. We begin by understanding how the web and HTTP work, then set up our development environment and build a minimal working server. From there, we add routing, middleware, templating, JSON

processing and database integration. We then tackle authentication, concurrency, memory management and security before covering testing, performance optimization, deployment and production maintenance.

Each chapter builds on the previous ones. You should read them in order rather than jumping ahead, because concepts and code examples accumulate. By the time you reach later chapters, you will have a complete working application that demonstrates everything discussed.

The book assumes you can read and write C. You should be comfortable with pointers, structures, memory allocation and basic system programming concepts. This is not a C tutorial. It is a web application development book that uses C as the implementation language. If you need to brush up on C fundamentals, spend some time with that first, then return to this material.

The Example Application Journey

Throughout the book, we build a single running example: a RESTful task management API that supports user authentication, project organization and task tracking. This application evolves incrementally, starting as a bare-bones HTTP server in Chapter 5 and gradually acquiring features until it resembles a production-ready service.

Starting from Chapter 5, our application begins as a simple TCP server that responds to HTTP GET requests with static content. By Chapter 6, it has a routing system that dispatches requests to handler functions based on URL paths. By Chapter 7, middleware provides cross-cutting concerns like logging and CORS handling. By Chapter 9, it processes JSON request bodies and returns JSON responses. By Chapter 10, SQLite provides persistent storage. By Chapter 12, users can register and authenticate. By Chapter 13, concurrent connections are handled efficiently through a thread pool.

The application is not a framework or a generic template. It is a concrete example that demonstrates patterns and techniques you can apply to your own projects. The code is organized into logical modules, uses a CMake build system, includes automated tests and follows conventions suitable for long-term maintenance.

All code examples are complete, compilable and runnable. You can copy them, build them and experiment with them. Where code is simplified for

clarity, this is explicitly stated so you understand what is missing for production use.

Development Environment and Conventions

The book assumes a Unix-like environment. All examples are tested on Linux with GCC or Clang and most concepts and code port to macOS and BSD systems with minor adjustments. Windows users can often adapt the code using MinGW, MSYS2, or WSL, but platform-specific behaviors in networking and system calls may require modifications.

For the C standard, we target C11, which is widely supported and provides features like atomic operations and improved threading support that are useful for concurrent server programming. The code is generally compatible with C17 as well.

We use CMake as our primary build system. CMake is the de facto standard for C projects requiring portability, library integration and test support. We also discuss Makefile-based builds where relevant, because understanding Make remains valuable for many development workflows.

Key libraries we use throughout the book include:

- cJSON for JSON parsing and generation
- SQLite for persistent data storage
- libargon2 for password hashing
- libjwt for JSON Web Token operations
- OpenSSL for TLS/HTTPS support
- libuv for cross-platform event-driven I/O (covered in Chapter 13)

Each library is introduced when first needed, with installation instructions and API overview. You do not need to install everything upfront. Follow along chapter by chapter and install dependencies as they arise.

Code in the book follows consistent naming and formatting conventions. Functions and variables use `snake_case`. Types use `CamelCase` or `snake_case` depending on context (system types and library types are kept as-is). Headers use a prefix matching their module to avoid naming conflicts. Files are named to match their primary function or module.

The project structure organizes source files into a `src/` directory, headers into an `include/` directory, tests into a `tests/` directory and configuration into a `config/` directory. Templates, static assets and build scripts live in their own directories. This structure grows with the application and mirrors conventions used in mature C projects.

With this foundation established, let us begin by understanding the protocols and architecture that underlie everything we will build. Chapter 1 explains how the web works, how HTTP requests travel from a browser to a server and how the HTTP protocol encodes the information our applications will parse and respond to.

Chapter 1: The Web, HTTP and How Requests Flow

When you type a URL into a browser and press Enter, something complex happens in the background. A chain of events begins that spans network interfaces, routing tables, protocol handshakes and ultimately your server application. Understanding this chain is not optional for a C web developer. You need to know what your server receives, in what format and under what conditions.

How a Browser Request Reaches Your Server

The journey starts with DNS resolution. The browser takes the domain name from the URL and asks a DNS resolver to translate it into an IP address. This may involve recursive queries across multiple DNS servers. When the IP address is found, the browser knows where to send data.

Next comes the TCP connection. The browser initiates a TCP three-way handshake with the server's IP address on port 80 (for HTTP) or port 443 (for HTTPS). The handshake consists of a SYN packet from the client, a SYN-ACK from the server and a final ACK from the client. Once complete, a reliable, ordered, bidirectional byte stream exists between browser and server.

If HTTPS is in use, a TLS handshake occurs over the established TCP connection. This involves exchanging certificates, negotiating a cipher suite and establishing shared encryption keys. After TLS completes, all data flowing over the connection is encrypted.

Finally, the browser sends the HTTP request as a sequence of bytes through the TCP stream. The server receives those bytes, parses them into an HTTP request structure, processes the request, constructs an HTTP response and sends the response bytes back through the same TCP connection.

The HTTP layer is what our C code interacts with directly. TCP delivers bytes; we turn those bytes into requests and responses. TLS encrypts and

decrypts those bytes. DNS gives us the destination address. Everything else is the web application layer.

HTTP/1.1 Message Structure

HTTP/1.1 defines a simple text-based protocol. Both requests and responses share the same basic structure: a start line, headers, an empty line and an optional body.

An HTTP request looks like this:

```
1 GET /api/tasks HTTP/1.1
2 Host: example.com
3 User-Agent: curl/7.88.1
4 Accept: application/json
```

The first line is the request line: METHOD SP URI SP HTTP-Version CRLF. The method specifies what the client wants to do. The URI identifies the target resource. The HTTP version indicates the protocol version.

A note on the transcripts in this book: every example is printed with plain line feeds because that is what your editor and terminal show, but the bytes on the wire are CRLF pairs. Your parser must look for CRLF, not for a bare line feed and it must not assume the two are interchangeable. A client that sends a bare LF where CRLF is required is speaking HTTP/0.9 and a server that accepts bare LF as a line terminator is accepting requests that intermediaries will reject.

Following the request line are zero or more header fields. Each header is a name-value pair on its own line, separated by a colon and space, terminated by CRLF (carriage return, line feed). Headers carry metadata about the request: which host it targets, what content types the client accepts, authentication credentials, cookies and more.

After the headers comes a single empty line (CRLF with no content). This blank line signals that headers are complete and that a body may follow.

If the request has a body, it appears after the empty line. For example, a POST request creating a task might look like this:

```
1 POST /api/tasks HTTP/1.1
2 Host: example.com
3 Content-Type: application/json
4 Content-Length: 50
5
6 {"title": "Write HTTP server in C", "priority": 3}
```

The Content-Length header tells the server exactly how many bytes to read for the body. This is critical for parsing. Without it (or without chunked transfer encoding), the server cannot know where the body ends.

HTTP responses follow a similar structure. The first line is the status line: HTTP-Version SP Status-Code SP Reason-Phrase CRLF. An example response to our task creation request:

```
1 HTTP/1.1 201 Created
2 Content-Type: application/json
3 Content-Length: 60
4 Location: /api/tasks/42
5
6 {"id": 42, "title": "Write HTTP server in C", "priority": 3}
```

The status code (201) indicates success with creation. The Reason-Phrase (“Created”) is human-readable but carries no semantic meaning. The headers describe the response metadata and the body contains the created task as JSON.

Parsing these messages in C means reading bytes until you find CRLF sequences, extracting the start line, parsing each header line into a name-value pair, detecting the blank line and then reading exactly Content-Length bytes for the body. This is straightforward in principle but requires careful handling of edge cases: partial reads, malformed headers, missing Content-Length and excessively large requests.

Methods, Headers, Status Codes and Semantics

HTTP defines a set of standard methods that indicate the intended action on a resource. GET retrieves a resource without side effects. POST creates a new resource or submits data for processing. PUT replaces an existing resource entirely. PATCH applies partial modifications. DELETE removes a resource. HEAD is identical to GET but returns only headers, no body. OPTIONS requests

information about available methods and headers and is used by browsers for CORS preflight checks.

Each method has specific semantic expectations. GET requests should be idempotent (repeating them produces the same result) and safe (no side effects). POST is neither safe nor idempotent. PUT and DELETE are idempotent: calling them multiple times has the same effect as calling them once. PATCH is idempotent when the patch operation itself is idempotent.

These semantics matter when designing RESTful APIs. A GET request that modifies data violates client expectations and breaks caching assumptions. A PUT request that appends instead of replacing is confusing and error-prone. Using methods correctly makes APIs intuitive and interoperable.

Headers carry crucial information. The Host header is mandatory in HTTP/1.1 and specifies which host the request is for. Content-Type declares the media type of the request body. Content-Length specifies body size. Accept tells the server what media types the client prefers. Authorization carries credentials. Set-Cookie allows the server to store state on the client. Cache-Control directs caching behavior. Connection controls whether the TCP connection should be kept alive after the response.

Status codes fall into five categories. 1xx codes are informational (like 100 Continue). 2xx codes indicate success (200 OK, 201 Created, 204 No Content). 3xx codes redirect the client (301 Moved Permanently, 302 Found, 304 Not Modified). 4xx codes indicate client errors (400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 405 Method Not Allowed, 429 Too Many Requests). 5xx codes indicate server errors (500 Internal Server Error, 502 Bad Gateway, 503 Service Unavailable).

Choosing correct status codes communicates meaning to clients and makes APIs easier to work with. Returning 200 OK for everything, with error information embedded in the response body, obscures failure modes and complicates client logic.

Connection Management and Keep-Alive

HTTP/1.1 introduced persistent connections by default. Instead of closing the TCP connection after each request-response pair, the connection stays open and can be reused for multiple requests. This dramatically reduces latency,

especially for pages that load many resources, because TCP handshake and TLS negotiation overhead is amortized across multiple requests.

Connection persistence is controlled by the Connection header. By default, HTTP/1.1 connections are kept alive unless either side sends “Connection: close”. When a server receives a request with Connection: close, it must close the TCP connection after sending the response. When a server sends Connection: close in the response, the client must not reuse the connection.

Implementing keep-alive in a C server means handling multiple requests over a single connection. After processing one request and sending the response, instead of immediately closing the socket, you read the next request. This requires careful state management: you must track whether the connection should persist, handle partial reads and writes correctly and detect when the client closes its end of the connection.

A simple strategy is to use a read loop that attempts to read the next request. When read returns zero, the client closed the connection. When read returns EAGAIN or EWOULDBLOCK on a non-blocking socket, no data is available yet. When read returns a negative value with errno set to ECONNRESET, the connection was forcibly closed. Each case requires different handling.

Timeouts are also important. A connection that sits idle, waiting for a request that never arrives, wastes resources. Implement a read timeout so that connections inactive for too long are closed and freed.

HTTP/2 and HTTP/3 at a Glance

HTTP/2 and HTTP/3 are binary protocol versions that improve on HTTP/1.1 in several ways. HTTP/2 multiplexes multiple requests over a single TCP connection, eliminating head-of-line blocking at the HTTP layer. It uses binary framing instead of text parsing, enabling more efficient encoding and transmission. It supports header compression with HPACK, reducing overhead for repetitive headers. The specification also defines server push, where the server sends resources before the client explicitly requests them, though you should be aware that push has been removed from every major browser and from current versions of curl and HTTP/3 obsoletes it outright. Treat push as a historical feature you will read about in older material, not something to design against.

HTTP/3 replaces TCP with QUIC, a UDP-based transport protocol that includes encryption and connection migration. QUIC provides multiplexing without head-of-line blocking at the transport layer, so a single lost packet does not stall all streams. Connection migration allows a client to change IP addresses without losing the connection, which is valuable for mobile devices switching between networks.

For the purposes of this book, we implement HTTP/1.1. HTTP/2 and HTTP/3 support in C is possible, but the complexity is significant. In production, the standard approach is to put a reverse proxy like Nginx or Caddy in front of your HTTP/1.1 server. The proxy handles TLS termination and speaks HTTP/2 or HTTP/3 to clients while communicating with your backend over HTTP/1.1. This lets you focus on application logic while still providing modern protocol support to end users.

Understanding HTTP/1.1 thoroughly is still essential, because HTTP/2 and HTTP/3 preserve the same request-response semantics, status codes and header structure. They change the transport and framing, not the application layer protocol.

Summary

HTTP is a simple protocol built on well-understood concepts: requests and responses, methods, headers, status codes and bodies. The text-based structure of HTTP/1.1 makes it parseable in C with straightforward string operations, though edge cases require careful handling. Connection management, timeouts and proper use of methods and status codes are foundational to building correct web applications.

With HTTP understood, the next step is setting up our development environment and establishing the project structure and build conventions we will use throughout the book. Chapter 2 covers compilers, standards, CMake configuration and modular design patterns for C web projects.

Chapter 2: C, Build Systems and Project Layout

Building a modern C web application requires more than writing code. It requires a coherent project structure, a reliable build system, consistent coding conventions and a strategy for managing dependencies. These are not aesthetic concerns. They are foundational decisions that determine whether your project grows into a maintainable system or collapses under its own complexity.

Compilers, Standards and Portability Choices

The two most widely used C compilers are GCC (GNU Compiler Collection) and Clang. Both support the major C standards and produce optimized machine code. GCC has been the default compiler on Linux distributions for decades and has broad feature support. Clang, developed by LLVM, offers faster compilation times, clearer error messages and strong integration with modern tooling like AddressSanitizer and static analyzers.

For this book, code is tested with both GCC 11+ and Clang 13+. The examples use no compiler-specific extensions and should compile on any standards-compliant C11 compiler.

The C standard we target is C11. C11 introduced several features useful for systems programming and concurrency: the `_Thread_local` storage class specifier, atomic types and operations in `<stdatomic.h>`, improved threading support via `<threads.h>`, the `_Generic` type-generic selection macro and `_Noreturn` (spelled `noreturn` via `<stdnoreturn.h>`) for functions that never return, such as the fatal-error helpers a server uses on startup. C11 is widely supported and represents a good balance between modern features and portability.

One consequence of choosing C11 deserves early warning, because it bites every later chapter. Compiling with `-std=c11` puts the compiler in strict ISO C mode and glibc then hides the POSIX and Linux APIs behind feature-test

macros. Sockets are declared, but `SO_REUSEPORT`, `accept4`, `epoll`, `sigaction` and `strdup` are not and you get “implicit declaration of function” errors on code that is otherwise correct. The fix is a single line in the build file, shown in the CMake configuration below and it is why that line is there rather than being an unexplained incantation.

C17 (ISO/IEC 9899:2018) was a bug-fix release over C11 with no new features; code that compiles under C11 compiles under C17 unchanged. The current standard is C23 (ISO/IEC 9899:2024), which does add real features: `true` and `false` as keywords, `nullptr`, standard `[[attributes]]`, `typeof`, the `#embed` preprocessor directive and the bounds-checking interfaces that were previously an optional annex. Current GCC and Clang implement most of C23 and `-std=c23` is available.

This book stays on C11 deliberately. C11 is what the compilers, CI images and long-lived distribution packages you are most likely to meet all agree on and nothing in the example application needs C23. Where a C23 feature would genuinely improve the code, the chapter that touches it says so.

Portability is a genuine concern for C web applications. If your application targets Linux servers, you can reasonably rely on POSIX APIs for sockets, threading and file I/O. If you need to support macOS, BSD, or other Unix-like systems, POSIX remains your friend, though some Linux-specific features like `epoll` will not be available (`kqueue` is the BSD equivalent). Windows is a different environment altogether; socket APIs differ, threading uses a different model and path conventions change. For Windows support, you would typically use `libuv` or a similar abstraction layer.

A practical approach is to design your application around POSIX APIs and isolate platform-specific code behind abstraction layers. This way, porting to a new platform means implementing a thin compatibility layer rather than rewriting the entire codebase.

Build Systems: Make, CMake and When to Use Each

Make is the classic Unix build tool. A Makefile describes targets (things to build), dependencies (what each target needs) and commands (how to build each target). Make determines which targets are out of date and runs the necessary commands to update them.

Make is simple and ubiquitous. Almost every Unix system has it installed. Writing a Makefile teaches you about the build process. For small projects or simple scripts, Make is perfectly adequate.

Where Make falls short is complexity. Large projects with multiple libraries, platform-specific configuration, test suites and dependency management quickly produce Makefiles that are fragile, repetitive and difficult to maintain. Make has no built-in cross-platform support. It assumes Unix-style paths, shells and command conventions.

CMake addresses these problems. CMake is a meta-build system: instead of building directly, it generates build files for native build tools. On Linux, CMake can generate Makefiles, Ninja build files, or project files for IDEs like CLion and Qt Creator. On Windows, it generates Visual Studio project files or MSBuild scripts.

CMake has become the de facto standard for C and C++ projects that care about portability. Libraries like OpenSSL, SQLite and libuv all use CMake. Modern CMake practices provide a clean, declarative way to describe targets, dependencies, compiler options and installation rules.

For our example application, we use CMake. This is not because Make is bad for everything, but because CMake scales better as the project grows, integrates with package managers and CI systems more naturally and makes dependency management cleaner.

Project Directory Structure and Modular Design

A good directory structure makes code discoverable, separates concerns and grows predictably. Our base structure looks like this:

```
1 task-api/
2 |— CMakeLists.txt
3 |— include/
4 |   |— task_api/
5 |       |— config.h
6 |       |— server.h
7 |       |— request.h
8 |       |— response.h
9 |       |— router.h
10 |      |— utils.h
11 |— src/
12 |   |— main.c
13 |   |— config.c
14 |   |— server.c
15 |   |— request.c
16 |   |— response.c
17 |   |— router.c
18 |   |— utils.c
19 |— tests/
20 |   |— CMakeLists.txt
21 |   |— test_request.c
22 |   |— test_router.c
23 |— config/
24 |   |— server.conf
25 |— docs/
26 |— README.md
```

The `include/` directory holds public headers. Headers are organized under a namespace directory (`task_api/`) to avoid naming conflicts. Each module has a corresponding header and source file: `request.h` and `request.c` implement HTTP request parsing, `router.h` and `router.c` implement URL routing and so on.

The `src/` directory holds implementation files. The `tests/` directory contains unit tests. A separate `CMakeLists.txt` in `tests/` keeps test build configuration isolated. The `config/` directory holds configuration files that are not compiled into the binary.

This structure scales. When we add middleware support, we create `middleware.h` and `middleware.c`. When we add authentication, we create `auth.h` and `auth.c`. Each module is self-contained with a clear public interface defined in its header file.

Modular design follows a few principles. Each module should do one thing. Each module should expose only what is necessary through its header. Implementation details belong in the `.c` file, not the header. Modules should depend on other modules through headers, not through implicit coupling like

shared global state or assumptions about execution order.

When designing the module interfaces, think about how they will be used. A good interface is small, focused and intuitive. Functions have descriptive names. Error handling is consistent. Data structures are opaque where possible, so implementation details can change without breaking callers.

Coding Conventions and Style for C Web Projects

Consistent style is not about aesthetics. It is about reducing cognitive load. When every file follows the same conventions, reading code becomes predictable and mistakes become easier to spot.

Naming conventions: functions and variables use `snake_case` (`handle_request`, `connection_count`). Types use `CamelCase` (`HttpConnection`, `RequestConfig`) or descriptive `snake_case`. Macros and constants use `UPPER_CASE` with underscores. Private functions and symbols that should not be used outside their module use a prefix indicating their module (`request_parse_header`, `router_match_route`).

Function signatures are clear and self-documenting. Parameters have descriptive names. Return types communicate intent: a pointer indicates a potential failure when `NULL` is returned, an `int` return typically indicates success or error codes.

Headers follow a defensive pattern. Include guards prevent multiple inclusion. Dependencies are explicit. Each header includes only what it needs. The corresponding `.c` file includes its own header first. This pattern, sometimes called “header-first include” or “including your own header”, ensures that each module compiles correctly on its own and that missing dependencies are caught early.

A typical header structure:

```

1  #ifndef TASK_API_CONFIG_H
2  #define TASK_API_CONFIG_H
3
4  #include <stddef.h>
5
6  #define DEFAULT_PORT 8080
7  #define DEFAULT_HOST "127.0.0.1"
8  #define CONFIG_PATH_ENV "TASK_API_CONFIG"
9  #define MAX_CONFIG_FILE_SIZE 65536
10
11  typedef struct Config {
12      char host[256];
13      int port;
14      int worker_threads;
15      char db_path[512];
16  } Config;
17
18  int config_load(Config *cfg, const char *path);
19  void config_defaults(Config *cfg);
20  void config_print(const Config *cfg);
21
22  #endif /* TASK_API_CONFIG_H */

```

Error handling uses a consistent convention throughout the codebase. Functions that can fail return negative error codes on error and non-negative values on success, or return NULL for failure when returning a pointer. Error codes are defined in a common header so that modules can return consistent values. Log messages accompany error returns to aid debugging.

Comments explain why, not what. The code itself should make it clear what is happening. Comments are for explaining non-obvious decisions, trade-offs and the rationale behind algorithms or design choices.

Installing Dependencies and Managing Libraries

C does not have a universal package manager like npm or pip. Library installation depends on your system and project needs. This section outlines practical approaches.

System package managers are often the easiest option for common libraries. On Debian-based distributions, packages like libssl-dev, libsqlite3-dev, libcmocka-dev and libargon2-dev are available through apt. On Fedora, they are available through dnf. These packages install headers and libraries into standard locations that the compiler and linker know about.

For libraries without system packages, or for controlled versioning, you have several options:

Download and build from source. Clone the library repository, configure with CMake (or autotools), build and install into a prefix directory like `/usr/local` or a project-local directory. CMake can then find the library using `find_package()`.

Use CMake's `FetchContent` module. This downloads and builds dependencies as part of your CMake configuration. It is clean, reproducible and avoids polluting the system with libraries. Our project uses `FetchContent` for some dependencies.

Use `vcpkg` or `Conan`. These are modern package managers for C and C++ that handle dependency resolution, downloading and building. They are more sophisticated than manual methods but add a layer of tooling to learn.

For simplicity, this book uses a combination of system packages and CMake `FetchContent`. Libraries with wide system support like `SQLite` and `OpenSSL` are installed via system package managers where possible. Libraries we need specific versions of, or libraries that are simpler to include directly, are managed via `FetchContent` or as vendored dependencies.

A practical note about header-only libraries: some libraries, like `cJSON`, can be included directly in your project as a single source and header file. This eliminates build-time dependencies entirely. For our project, we will sometimes use this approach for small, stable libraries to keep the build simple and self-contained.

A Minimal CMake Configuration

Let us look at the `CMakeLists.txt` that builds our example application. This configuration grows as we add features throughout the book. The initial version handles the core server executable:

```

1  cmake_minimum_required(VERSION 3.16)
2  project(task_api C)
3
4  set(CMAKE_C_STANDARD 11)
5  set(CMAKE_C_STANDARD_REQUIRED ON)
6  set(CMAKE_C_EXTENSIONS OFF)
7
8  # We stay in strict ISO C11 mode, so glibc hides the POSIX and Linux APIs this
9  # project calls behind feature-test macros: SO_REUSEPORT and accept4, epoll,
10 # sigaction, strdup. Ask for them explicitly, in the build file, where a
11 # reader can see them. Without this line the socket code in Chapter 3 and the
12 # event loop in Chapter 13 fail to compile with -std=c11.
13 add_compile_definitions(_GNU_SOURCE)
14
15 # Build type defaults to Release if not specified
16 if(NOT CMAKE_BUILD_TYPE)
17     set(CMAKE_BUILD_TYPE Release)
18 endif()
19
20 # Common compiler warnings. Guarded by compiler id, because these flags are
21 # GCC/Clang spellings and MSVC rejects them outright.
22 if(CMAKE_C_COMPILER_ID MATCHES "GNU|Clang")
23     add_compile_options(-Wall -Wextra -Wpedantic)
24 elseif(MSVC)
25     add_compile_options(/W4)
26 endif()
27
28 # Find required packages
29 find_package(SQLite3 REQUIRED)
30
31 # Define the main executable
32 add_executable(task_api
33     src/main.c
34     src/config.c
35     src/server.c
36     src/request.c
37     src/response.c
38     src/router.c
39     src/utils.c
40 )
41
42 # Include directories. Headers live under include/task_api/, so a source file
43 # reaches them as #include "task_api/config.h" -- the namespace directory is
44 # part of the include path, not something to flatten away.
45 target_include_directories(task_api
46     PRIVATE
47         ${CMAKE_SOURCE_DIR}/include
48         ${SQLite3_INCLUDE_DIRS}
49 )
50

```

```

51 # Link libraries
52 target_link_libraries(task_api
53     PRIVATE
54         ${SQLite3_LIBRARIES}
55         pthread
56 )
57
58 # Testing. include(CTest) provides the BUILD_TESTING option and calls
59 # enable_testing(); without it the tests/ directory the project layout promises
60 # is never configured and ctest reports no tests.
61 include(CTest)
62 if(BUILD_TESTING)
63     add_subdirectory(tests)
64 endif()
65
66 # Installation
67 install(TARGETS task_api DESTINATION bin)
68 install(DIRECTORY config/ DESTINATION etc/task_api)

```

This configuration sets the C standard to 11, enables compiler warnings, finds SQLite, defines the executable with its source files, specifies include directories, links required libraries and defines installation targets. The PRIVATE keyword on include directories and link libraries indicates that these dependencies are needed only for building task_api, not for any library that might depend on it.

Three lines in there are load-bearing in ways that are easy to dismiss as boilerplate.

The add_compile_definitions(_GNU_SOURCE) line is the feature-test macro discussed above. Delete it and the project stops compiling at Chapter 3, with errors that point at the socket code rather than at the build file, which is exactly the kind of failure that costs an afternoon.

The warning flags are guarded by CMAKE_C_COMPILER_ID. The book targets GCC and Clang and the prose promises that CMake is what buys you portability, so the flags have to be spelled per compiler or that promise is false the moment someone opens the project in Visual Studio.

The include(CTest) block is what makes the tests/ directory in the project layout real. A tests/CMakeLists.txt that the top-level build never adds is configured by nobody and run by nothing; ctest reports no tests and the suite silently does not exist. include(CTest) gives you the BUILD_TESTING option for free, so a production build can be configured with -DBUILD_TESTING=OFF.

Note also the include path. The layout puts headers at `include/task_api/-config.h` and the build adds only `include/`, so every source file in this project writes `#include "task_api/config.h"`. The namespace directory is part of the include path, not a detail you can flatten. Source files in this book use that spelling throughout.

As we add features, we will enhance this configuration to include JSON libraries, testing and additional dependencies. The structure remains the same: find dependencies, define targets, specify sources and dependencies, configure build options, define installation.

Summary

A solid foundation in build tools, project organization and coding conventions is essential for any non-trivial C project. CMake provides a scalable build system. A clear directory structure with modular separation of concerns keeps code organized as the project grows. Consistent naming and style conventions reduce cognitive load and improve code quality.

With our development environment established, we can now focus on the networking layer. Chapter 3 dives into POSIX sockets: creating TCP servers, accepting connections, handling blocking and non-blocking I/O and managing the lifecycle of client connections. This is the plumbing that makes web servers work.

Chapter 3: Sockets and Network Programming in C

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Creating and Configuring a TCP Server Socket

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Accepting Connections and Handling Clients

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Blocking, Non-Blocking and Interrupted Syscalls

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Reading and Writing: Buffer Handling and Partial Operations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Graceful Shutdown and Connection Cleanup

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 4: Parsing HTTP Requests

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Designing a Request Parser

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Reading Until CRLF: Line-Based Parsing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Parsing Headers into a Structured Map

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Handling the Message Body and Content-Length

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Error Handling and Malformed Requests

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 5: Building a Minimal Web Server

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Server Initialization and the Main Loop

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Request Handling and Response Generation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Serving Static Content from Disk

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Returning JSON Responses

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Testing the Server with curl

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 6: Routing and URL Dispatch

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Route Registration and Handler Signatures

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Exact Matches, Parameter Extraction and Wildcards

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Trie-Based Route Matching for Performance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

404 Handling and Method Not Allowed

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Organizing Handlers Across Modules

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 7: Middleware and Request Processing Pipelines

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

The Middleware Pattern and Execution Flow

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Writing Logging and Timing Middleware

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

CORS Handling

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Error Propagation and Short-Circuiting

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Composing Middleware Chains

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 8: Templating and Dynamic HTML

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Server-Side Rendering in C

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Using Mustache or a Similar Template System

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Template Loading and Caching

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Escaping and Preventing Injection

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Generating Complex Pages from Data

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 9: JSON Processing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Choosing a JSON Library: cJSON vs Jansson

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Parsing JSON Request Bodies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Building JSON Responses Programmatically

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Error Handling and Validation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

JSON Schema and Request Validation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 10: Database Integration with SQLite

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Why SQLite for C Web Applications

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Opening Databases and Connection Lifetimes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Prepared Statements and Parameter Binding

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Transactions and Concurrency

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Schema Design and Migration Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 11: Building RESTful APIs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

REST Principles and Resource Design

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

CRUD Operations and HTTP Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Pagination, Filtering and Sorting

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

API Versioning Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Error Responses and Consistent Envelopes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 12: Authentication and Authorization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Authentication vs Authorization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Password Hashing with Argon2 or bcrypt

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Session-Based Authentication

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

JWT and Stateless Tokens

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Role-Based Access Control Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 13: Concurrency and Asynchronous I/O

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Threading Models: Thread-Per-Request vs Thread Pools

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Implementing a Worker Thread Pool

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Event Loops and Non-Blocking I/O with epoll/kqueue

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Using libuv for Cross-Platform Evented I/O

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Choosing the Right Concurrency Strategy

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 14: Memory Management and Resource Safety

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Ownership Semantics and Transfer Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Arena Allocation for Request-Scoped Data

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Reference Counting and Shared Resources

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Detecting Leaks with Valgrind and ASan

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Defensive Coding Against Buffer Overflows

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 15: Error Handling, Logging and Observability

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Designing Error Codes and Error Propagation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Centralized Error Handling in the Request Pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Structured Logging with JSON Output

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Log Levels, Correlation IDs and Debugging

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Integration with Log Aggregation Tools

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 16: Configuration, File Uploads and Caching

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Loading Configuration from Files and Environment

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Parsing and Validating Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Handling Multipart File Uploads

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

In-Memory Caching Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Response Caching and Cache-Control Headers

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 17: Secure Coding and Web Vulnerabilities

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Input Validation and Sanitization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Cross-Site Scripting (XSS) Prevention

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

SQL Injection Defense

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Buffer Overflow and Use-After-Free Mitigations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Secure Cookie Handling and Headers

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 18: TLS, HTTPS and OpenSSL

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Why TLS Is Non-Negotiable

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Setting Up Certificates and Private Keys

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

TLS Handshake and Cipher Suites

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Integrating OpenSSL with the Server

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

HTTP to HTTPS Redirection

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 19: Testing and Debugging C Web Applications

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Unit Testing Frameworks: CMocka and Unity

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Writing Tests for Handlers and Routes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Integration Testing Against a Running Server

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Property-Based Testing and Fuzzing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Debugging Techniques and Tools

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 20: Performance Optimization and Profiling

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Measuring Performance: Benchmarks and Metrics

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Profiling with perf, gprof and Flame Graphs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Reducing Allocation and Copying

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Efficient I/O and Buffering

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Latency vs Throughput Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 21: Deployment Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Reverse Proxies: Nginx and Caddy

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Process Management with systemd

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Static vs Dynamic Linking for Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Reverse Proxy Configuration for Your Server

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Load Balancing and Multi-Instance Deployments

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 22: Containerization and CI/CD

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Dockerfile Best Practices for C Applications

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Multi-Stage Builds for Smaller Images

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

GitHub Actions and CI Pipelines

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Automated Testing in CI

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Deployment Strategies and Rollbacks

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Chapter 23: Monitoring and Production Maintenance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Health Checks and Readiness Probes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Metrics Export and Prometheus Integration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Structured Logging at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Handling Incidents and Debugging in Production

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Updates, Hotfixes and Rolling Deployments

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Conclusion: Beyond the Book

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

The Journey So Far: From Socket to Production

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Advanced Topics Worth Exploring

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

The Future of C and Web Development

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Continuing Your Practice

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Appendix A: Glossary of Essential Terms

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Appendix B: Quick Reference

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Essential Build Commands

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Common HTTP Status Codes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Project Structure

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Security Checklist

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

References

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

HTTP and Web Standards

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Libraries and Tools

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Build Systems

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Security

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Nginx and Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

GitHub Actions and CI/CD

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Prometheus and Monitoring

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Performance and Benchmarks

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.

Additional Resources

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/buildingmodernwebapplicationsinc>.