DIGITAL EDITION · ENGLISH

The new technical *creator*.

FREE SAMPLE

Introduction · Complete contents · Chapter 2

AUTHOR

Hernán Capucci

Independent edition · 2026

This sample includes the introduction, the complete table of contents, and one representative chapter from the book. The goal is to let you evaluate the tone, the approach, and the kind of learning before getting the complete edition.

What this sample includes

- The complete introduction (Chapter 0).
- The complete table of contents.
- Chapter 2 in full: *How the internet works*.
- Continuity information.

Complete contents

Chapter 0 — Why this book exists

One day you ask AI to generate something for you — code for a feature, an automation connecting two services, a snippet that solves a specific problem. What it gives back seems to

work. You use it. And at some point in that process, a question appears that you may not dare ask out loud: how do I know if this is right?

That question is not a small one. It's the difference between operating with your own judgment and operating while hoping nobody finds the mistake.

This book exists so you can answer it.

Access without understanding

A few years ago, building **software** required specific technical training. Learning a programming language, understanding data structures, configuring development environments, reading documentation that assumed you already knew a lot. The barrier was high and real.

Then came **generative artificial intelligence** tools. Suddenly, anyone could write an instruction in natural language and receive working code, a configured automation, a system connected to external services. The technical barrier dropped significantly.

But what dropped was the technical barrier, not the conceptual one.

A tool generating code for you doesn't mean you understand what it generated. A platform deploying your application with one click doesn't mean you understand what a deployment is, what can fail, or how to roll back when something goes wrong. AI explaining an error doesn't mean you know why it happened or how to keep it from happening again.

Access improved. Understanding of the **system** underneath, in many cases, didn't keep pace.

Creating is not the same as understanding

Today there are more people building digital products without technical training than at any point before. Apps, automations, integrations between services, content platforms, digitized internal processes.

That democratization is real, and it's valuable. The problem appears when something doesn't work as expected, or when decisions have to be made about the system that was built.

An entrepreneur who used AI to build his online store doesn't know why "the database went down." He doesn't know whether his customers' information is protected. He doesn't know whether the work the developer he hired delivered is well done or not. He doesn't know what questions to ask to find out.

A professional who automated her processes with AI doesn't understand precisely what each part of what she assembled does. She accepts the result because it worked yesterday. When it stops working, she has no tools to figure out where to look.

A founder who works with an external technical team nods when they talk about **APIs**, **repositories**, or system architecture. But she doesn't understand. And she doesn't ask, because she doesn't know what to ask.

In all these cases the problem is not a lack of intelligence or ability. It's the lack of the vocabulary and the conceptual framework that make it possible to operate in the digital ecosystem with your own judgment.

AI as the starting point

Artificial intelligence is what made this book urgent. It's the wake-up call.

But the book's destination is not artificial intelligence. It's the systems that exist beneath any digital project, with AI or without it.

An API is not an AI concept. It has existed for decades. A code **repository** isn't new either. The difference between a development environment and a production environment wasn't invented by any language model. User authentication, relational databases, the life cycle of a **deploy** — all of this existed before generative AI and will keep existing after it.

What changed with AI is that these things are now within reach of people who never touched them before. And that accessibility without understanding creates new dependencies.

Dependency on the tool: if it changes, if it fails, if it produces unexpected results, you have no way to evaluate them. Dependency on the developer who does understand: if he changes, if he raises his rates, if he disappears, you can't continue without him. Dependency on whoever claims to know more: without a framework of your own, you can't tell real knowledge from empty promises.

AI opened a door. This book gives you the vocabulary to understand what's inside.

The missing vocabulary

There's a frequent confusion worth clearing up from the start.

The problem this book addresses is not that you don't know how to program. Not knowing how to program is not the problem. An enormous number of people who are highly effective in the digital world don't know how to program and don't need to learn.

What matters is not knowing how to write code. What matters is knowing how to read it with enough judgment to understand what it does. What matters is understanding what a **database** is so you can make decisions about how to store information. Knowing what a repository is so you can review what a developer delivered. Knowing what an **endpoint** is so you can ask AI to build one with precision. Knowing what a production environment is so you don't break something live by mistake.

None of that requires writing a single line of code.

Picture this specific situation: the developer tells you "the migration failed in the staging environment." Without context, that sentence is noise. With the vocabulary in this book, you understand that a "migration" is a change to the structure of the database, that "staging" is the testing environment that comes before production, and that the problem probably hasn't affected users yet — but it has to be resolved before moving on. That difference doesn't require programming. It requires understanding the concepts.

What it does require is technical vocabulary. The vocabulary that lets you ask the right questions, interpret answers, evaluate proposals, spot warning signs, and make decisions with judgment. That vocabulary isn't acquired by using tools. It's acquired by understanding the concepts behind them.

That's what this book builds. Concept by concept, from zero, with universal examples and without assuming any prior knowledge.

Who this book is for

This book is written for people who build digital things without technical training and want to do it with better judgment.

That includes whoever is putting together their first digital product with the help of AI and doesn't fully understand what they're building. Whoever works with an external technical team and wants to talk with them without every conversation being an exercise in translation. Whoever uses automation tools in their daily work and wants to understand what they're

actually doing. Whoever has an idea and needs to evaluate it technically before committing time and money to developing it.

The industry doesn't matter, and neither does the field of work. Age and level of formal education in technology don't matter either.

What does matter is that you use a computer fluently in your work or project. That you've tried at least one AI tool. And that you want to understand the **digital ecosystem** you operate in better.

Your starting point might be that you've never opened a code repository. That you've never configured a development environment. That you're not sure what the difference is between a server and a cloud. None of those starting points is an obstacle. This book starts from there.

What you won't find

Before going on, it's worth being explicit about what this book doesn't do.

It doesn't teach programming. It's not a manual for any programming language or any specific tool. If that's what you're looking for, there are better and more specialized resources for it.

It doesn't claim that AI does everything. It doesn't. It has real capabilities and real limits. This book shows both honestly.

It doesn't promise that your project will be ready within some fixed timeframe. Technical processes have their own complexity and their own pace. Minimizing them would be lying.

It doesn't depend on any specific tool, platform, or service. When a concrete tool is mentioned, it's as an example. The concepts taught here apply equally with whatever stack or environment you use.

It doesn't discuss particular success stories or real projects as models to follow. The examples are generic and universal by design.

It doesn't say "it's easy" when it isn't. Some concepts in this book are simple. Others demand attention and a second read. When something has real complexity, the book says so.

What you will find

By the end of this book, you'll have a concrete understanding of concepts you may only know by name today.

You'll understand what an application is on the inside: which part handles what you see on screen, which part processes the logic and the data, and how those two parts communicate. That won't make you able to build it alone, but it will mean you understand what people are talking about when they describe it to you, or when something fails.

You'll be able to open a code repository and read its structure: what folders exist, what each part of the project does, what the change history says. That information is available in any repository, and today you may not know you can use it to understand what someone delivered to you.

You'll be able to write a **prompt** with enough context for AI to produce something useful. The difference between a vague instruction and a precise one is not talent: it's knowing what information the tool needs to do its job well.

You'll be able to review what AI or a developer generated with concrete questions: what exactly does this part do? What happens if this field arrives empty? Is there any section that accesses sensitive information? Those questions don't require knowing how to program. They require understanding the concepts at play.

You'll be able to use this book's glossary as a working tool: open a definition when an unfamiliar term appears, understand it, and return to the conversation or the document with more clarity than before.

And you'll be able to recognize when an answer — from a person or from a tool — can't be evaluated with your current knowledge. That doesn't always mean knowing the right answer. It means knowing when the question you need to ask is better than the one you asked.

How it's organized

The book has six parts plus this introductory chapter, practical appendices, and a technical glossary.

Part I covers the foundations: what software is, how the internet works, what the client-server model is, what a database is, and what code is. These are the most basic concepts of the digital ecosystem. Every part that follows takes them as known.

Part II goes into the architecture of an application: the **frontend**, the **backend**, APIs, databases in depth, authentication and sessions. It's the internal structure of any modern digital product.

Part III covers the tools and processes that make a project work in the real world: version control with **Git**, the command-line terminal, development environments, deploys, and basic security.

Part IV is dedicated to working with AI intelligently: what it can do, where it fails systematically, how to write an effective prompt, how to review what it produces, and how to use models, **agents**, and automations without losing control of the process.

Part V brings it all together in the context of building something concrete: how to prepare before starting, how to communicate with technical teams, how to go from an idea to a working product.

Part VI is the practical appendices: quick-reference checklists, basic terminal and Git commands, reusable prompt templates, and resources for going deeper in each area.

At the end of the book there's a **technical glossary** with more than 200 terms explained in clear language, with examples and without jargon. It's not a decorative appendix. It's an integral part of the book.

The order of the parts is not arbitrary. Each one builds on the previous. The systems and tools parts need the foundations of Part I. The AI and building parts need the architecture and the tools. If you choose to read in order, that's why.

How to use it

There are two ways to read this book, and both are valid.

The first is straight through, from beginning to end. It's the recommended path for anyone with little or no prior contact with the digital ecosystem. The journey builds understanding cumulatively: each chapter prepares the ground for the next.

The second is as a reference book. Someone mentioned an API and you didn't understand what it is: you open Chapter 8. You're about to do your first deploy and don't know what it involves: you open Chapter 14. You need to understand what **authentication** is before a meeting about security: you open Chapter 10. Each chapter can be read independently, though cross-references point out what's worth having read first.

The glossary works the same way. When a technical term appears for the first time in a chapter, it's in **bold** with a reference to the glossary. In the digital version of the book, that reference is an active hyperlink: you can go straight to the definition and return to the chapter with one click. In the print version, the reference indicates the page number.

A note about the examples: throughout the book you'll find the same kinds of contexts — an online store, an appointment-booking app, a course platform. They're generic situations chosen because they require no prior knowledge of any industry. The concepts apply exactly the same in any other type of project.

A note about the tools: when a tool is mentioned as an example, it's just that, an example. The concepts this book teaches don't depend on any specific platform. If the tool you use today changes tomorrow, the concepts remain valid.

Creating with your own judgment

There's a difference between someone who uses digital tools without understanding them and someone who uses them knowing what they're doing.

That difference is not in knowing how to program. It's in having the vocabulary to ask the right questions. The conceptual framework to evaluate answers. The ability to detect when something is wrong even if you don't know exactly why. The autonomy to make decisions without depending completely on others to understand what's going on.

That's the profile this book builds: the new technical creator. Someone who uses AI, works with technical teams, builds digital things — and does it with their own judgment.

Not a programmer. Not trying to be one. But not operating blindly either, not accepting results they can't evaluate, not depending on someone else to explain everything.

They understand the system they work in. They know what to ask, how to evaluate what they receive, and when to recognize that something isn't right. That understanding doesn't require writing code. It requires understanding how the things you use actually work.

What you learned in this chapter

- AI democratized access to technical execution, but not understanding of the digital ecosystem.
- Not knowing how to program is not the problem. Lacking a minimal technical vocabulary is.
- This book doesn't teach programming. It teaches the concepts that let you create with judgment, talk with technical teams, and review what AI produces.
- The book has six parts, practical appendices, and a glossary of more than 200 terms. It can be read straight through or used as a reference.
- In the digital version, glossary terms are active hyperlinks in every chapter.

What comes next

Chapter 1 starts from the beginning: what software is and why understanding it matters even if you'll never write it. It's the first block of Part I and the foundation of everything that comes after.

Chapter 2 — How the internet works

You type an address into the browser and press Enter. In less than a second, a page appears: text, images, up-to-date data.

Where did all of that come from? How did it reach your screen? Why does it sometimes fail to arrive?

Answering those questions doesn't require knowing how to program. It requires understanding the infrastructure that connects any device to any server in the world. That understanding is the foundation for diagnosing why something works — and why sometimes it doesn't.

The journey of a request

When you type an address into the browser, what you're typing is called a **URL** — Uniform Resource Locator. It's the address that identifies a specific resource on the internet: a page, an image, a file, a piece of data.

The browser needs to find the server that has that resource. But servers aren't identified by readable names: they're identified by numbers. Every server has an **IP address** — a string of numbers that locates it uniquely on the network, much like a postal address locates a building in a city.

The problem is that you don't type numbers. You type `www.encyclopedia.org` or `news.example.com`. For the browser to find the corresponding server, it needs to translate that name into an IP address. That process is called **DNS** resolution — Domain Name System.

DNS works like the internet's phone book. The browser queries a DNS server and asks: "what IP does `news.example.com` have?" The DNS server answers with the number. The browser now has the real address.

With that address, the browser sends a request to the server. That request travels across the network — through cables, optical fiber, and wireless connections — until it reaches the server waiting for it. The server receives it, processes whatever it needs to process, and returns a response. That response travels back to the browser. The browser interprets the content and displays it on screen.

That whole process — DNS lookup, trip out, processing, response back — happens in fractions of a second under normal conditions.

One optimization worth mentioning: the **cache**. The browser doesn't query DNS every time you visit the same site. It saves the answer — the name-to-IP translation — for a while. The operating system does the same. That makes repeat visits faster. But it also means that if the server recently changed its IP address, your device may still be using the old one. Clearing the browser cache or waiting for it to expire solves that problem.

When you type an address, the browser doesn't go to a magical cloud: it starts a chain of translations, requests, and responses.

What a domain is

A **domain** is the readable name that identifies a site or service on the internet. `encyclopedia.org`, `news.example.com`, `myapp.io` are domains.

Domains don't exist on their own: someone registers them. There are companies that specialize in registering domains — they're called registrars — and a domain is rented for periods of

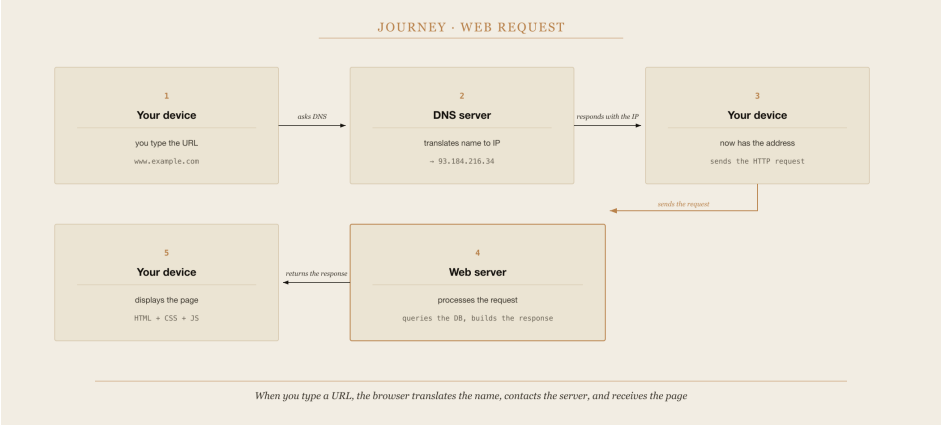

Figure 1. When you type an address, the browser doesn't go to a magical cloud: it starts a chain of translations, requests, and responses.

time, usually a year. If it isn't renewed, the domain expires and becomes available for someone else to register.

The final part of the domain — `.com`, `.org`, `.io`, `.ar` — is called the top-level domain, or TLD. It indicates the type of organization or the country of origin. That distinction has historical and positioning value, but technically it doesn't change how the system works.

The part that can be confusing is the relationship between domain and server. A domain is not the server: it's the label that points to the server. The same server can have several domains pointing to it. The same domain can redirect to different servers depending on the time of day or the region of the world you're connecting from. DNS is the system that keeps that relationship between names and addresses up to date.

When a domain isn't configured correctly — or when it has expired — DNS can't resolve the address. The browser doesn't know which server to go to. The page doesn't load, even if the server holding it is working perfectly.

What a server is — and what hosting is

A **server** is a computer designed to receive requests and return responses. At its core, it's no different from any other computer: it has a processor, memory, storage, and a network connection.

The practical difference is that a server is on permanently, connected to the internet with high availability, and configured to handle multiple simultaneous requests. It has no screen

or keyboard. It's not on anyone's desk. It lives in a data center — a facility designed to keep many servers running continuously and securely.

Hosting is the service that rents you space and resources on those servers. When you "publish" something on the internet — a page, an application, a file — what you're doing is placing that content on a hosting company's server, configuring a domain to point there, and letting the server deliver it to whoever requests it.

There are different forms of hosting. In the most basic one, you share the server with other sites — cheaper, but with limited resources. In more advanced forms, you have a dedicated virtual or physical server — more expensive, but with greater control and capacity. In the cloud model, resources are allocated dynamically based on demand.

For someone building something digital, understanding hosting means understanding that the system you created lives in a physical place, that this place has capacity limits, and that if it fails — or if the domain doesn't point to it correctly — the system isn't accessible even if it's perfectly built.

A useful distinction: the web server is the component that receives HTTP requests and returns content. The application server is the one that runs the system's logic — which Chapter 3 covers in more detail. In many simple systems, both functions run on the same machine. In larger systems, they're separated. What matters for now is that "the server" is not one monolithic thing: it's a set of pieces working together to answer requests.

HTTP and HTTPS: the language of the web

When the browser sends a request and the server returns a response, both need to speak the same language. That language is called a **protocol**. A protocol is a set of rules that defines how messages are formatted, transmitted, and interpreted.

The protocol of the web is **HTTP** — HyperText Transfer Protocol. It defines the structure of requests and responses: what information goes first, how the content type is indicated, how the result of the operation is communicated.

A basic HTTP request includes: the method (what you want to do — get information, send data, delete something), the address of the resource, and headers carrying additional information about who's asking and what kind of response they accept. An HTTP response includes:

the status code (whether it worked or not), headers with information about the content, and the body — the content itself.

HTTPS is the secure version of HTTP. The "S" stands for "Secure." The technical difference is that in HTTPS all communication travels encrypted: the data the browser sends to the server and the data the server returns can't be read by anyone intercepting the connection along the way.

That encryption is made possible by an **SSL/TLS certificate** — a digital file the server presents to the browser to prove it is who it claims to be, and which activates the encryption of the communication. Modern browsers show a padlock in the address bar when the connection uses HTTPS.

Why it matters for someone who builds: any system that handles user data — passwords, personal information, payments — must use HTTPS without exception. Without HTTPS, that data travels across the network in plain text and can be captured. With HTTP, the browser may show a "not secure" warning that drives users away and erodes trust in the system.

> ***Key concept: protocol*** *A set of rules that defines how two parties communicate. HTTP is the protocol browsers and servers use to exchange information on the web. HTTPS is its encrypted version.*

What the server answers

Every time a server receives a request, it returns a response that includes a number: the **status code**. That number indicates what happened to the request.

The codes are grouped by category. Those starting with 2 mean success. Those starting with 3 are redirects. Those starting with 4 indicate an error on the client side — something in the request is wrong. Those starting with 5 indicate an error on the server side — the server received the request but couldn't process it correctly.

The three most important ones for anyone working with systems:

200 — OK. The request was processed correctly and the server returned the expected content. It's the normal result. If you see a page, the server answered 200.

404 — Not Found. The server received the request, but the resource you asked for doesn't exist on that server. The address you used doesn't correspond to any available file, page, or data. It's not a server error: it's that what you were looking for isn't there.

500 — Internal Server Error. The server received the request, tried to process it, and something failed in its own workings. The problem isn't in what you asked for: it's in how the server handled the request. It's an error in the code or in the server's configuration.

Code	Name	What happened	Everyday example
200	OK	The server found and delivered what you asked for.	You open a page and everything loads without problems.
301 / 302	Redirect	The resource is at another address. The browser takes you there automatically.	An old URL sends you to the site's new domain.
400	Bad Request	The request arrived malformed; the server couldn't interpret it.	You submit a form with required fields incomplete.
401	Unauthorized	You didn't present valid credentials. The server doesn't know who you are.	You try to view private content without logging in.
403	Forbidden	Your credentials are valid, but you don't have permission for that.	You're logged in, but that section doesn't match your role.

Code	Name	What happened	Everyday example
404	Not Found	What you asked for doesn't exist on that server.	A URL was mistyped or no longer exists.
500	Internal Server Error	The server received your request but failed while processing it.	You try to complete a purchase and an "unexpected error" appears.
503	Service Unavailable	The server can't respond right now; it's overloaded or under maintenance.	A site goes down during a massive sale or launch.

There's no need to memorize every code. What matters is recognizing the pattern: 2xx usually means success, 3xx redirection, 4xx a problem on the request side, and 5xx a problem on the server side.

Why the app "went down"

"The page won't load," "the system is down," "it's not working" — when something stops being available, there's usually a specific technical cause. Understanding the possible points of failure lets you narrow down the problem before calling someone to fix it.

The most common causes, ordered by where they occur:

The server is down. The server stopped responding. It could be because the hosting service had a problem, because the server restarted for an update, or because the system ran out of resources (memory, CPU) and stopped working. In this case, the browser receives no response of any kind.

The domain doesn't resolve. DNS can't translate the domain name into an IP address. It could be because the domain expired, because the DNS configuration was changed incorrectly, or because there's a problem with the DNS server. The result is similar to the previous case: the browser doesn't know where to go.

The certificate expired. If the server's HTTPS certificate expired, the browser blocks the connection and shows a security warning. The page technically exists and the server works, but the browser protects the user by rejecting the connection. The visible result: a warning screen, not the page.

Error 500. The server responds, but with an internal error. The page exists, the server is on, but the code hit a problem while processing the request. The user sees an error message instead of the expected content.

Network problem. The connection between the user's device and the server is interrupted at some intermediate point. It could be the user's internet connection, a network provider, or a point in the infrastructure between the two sides.

Telling them apart, even superficially, helps you know where to start:

- If the page loads nothing and there's no warning → possibly a downed server or unresolved DNS.
- If the browser shows an explicit security notice → expired certificate or unencrypted HTTP.
- If the page loads with an error message in the content → probably error 500, a code problem.
- If only you can't get in but others can → possibly a network problem or local cache issue.

Knowing this doesn't mean being able to fix it yourself. It means being able to communicate it precisely to whoever can.

> ***Useful questions when a site won't load***
>
> *When the system isn't available, these questions help narrow down the problem before calling whoever can fix it:*
>
> - *Is the problem happening to everyone or just to me?*
> - *Does the domain resolve? Is the name pointing to any address?*

> - *Does the server respond? Is anything coming back, even an error?*
> - *What error code is there — 404, 500 — or is there no response at all?*
> - *Is production failing, or just a test environment?*
> - *Are there logs of the error?*
> - *Was there any recent change — a deploy, a configuration change, a domain renewal?*

There's one case worth mentioning because it's frequent and confusing: the system works for some people and not for others. That almost never means the server is down. It usually indicates a DNS propagation issue — when a change was made to the domain configuration, that change takes between minutes and hours to reach every DNS server in the world. During that time, different users query different DNS servers and get different answers. Some see the new site; others still see the old one, or nothing at all. It's expected behavior, not an error to fix urgently.

How you'll hear it in a meeting

In a meeting with a technical team or a vendor, people use phrases that assume everyone understands what's being discussed. These are the most common ones on this topic, what they mean in practice, and what you can ask.

"Looks like a DNS problem." The domain name isn't resolving correctly — it can't be translated into the server's IP address. It could be a misconfiguration or a recent change that hasn't fully propagated yet.

Useful question: "Is it happening to all users or only some?" If it's everyone, it's probably a configuration error. If it's only some, it's most likely propagation in progress — a process that takes hours and resolves on its own.

Don't assume: that the server is down. DNS and the server are separate pieces. The server can be working perfectly while DNS fails.

"The server is returning 500." The server received the request but hit an error while processing it. The problem is in the code or in the system's configuration, not in the user's request or the network.

Useful question: "Are there logs of the error?" Logs are the record of what the system did before failing. A 500 almost always leaves a trace that identifies the cause. Without logs, diagnosis becomes much harder.

Don't assume: that it's an internet problem or the user's device. A 500 means the server actually responded — just with an internal error.

"The certificate expired." The server's HTTPS certificate expired. The browser detects that the connection can no longer be guaranteed as secure and blocks access by default. The system may be working perfectly on the inside, but no user can get in.

Useful question: "How long does the renewal take?" In most cases it's a quick process. What matters is the estimated time to resolution and whether there are affected users who need to be informed.

Don't assume: that there's a code or infrastructure problem. It's an administrative expiration, much like a document expiring. It says nothing about the quality of the system.

"It's down in production." The system real users rely on is not available. It's not a test or development environment: it's the one with the real traffic and the real data. This phrase signals urgency.

Useful question: "Is someone already working on it? What's the estimated time?" Those two questions give you the context you need to decide whether to tell users something or wait quietly.

Don't assume: that the problem affects everyone equally. Sometimes the outage is partial — affecting only one region, one device, or one specific feature.

What you learned in this chapter

- A URL is the address that identifies a resource on the internet. To reach it, the browser queries DNS, which translates the domain name into an IP address.
- A server is an always-on computer that receives requests and returns responses. Hosting is the service that rents that server.
- HTTP is the protocol that defines how the browser and the server communicate. HTTPS is its encrypted version, mandatory for any system that handles sensitive data.
- Status codes are the language the server uses to say what happened: 200 is success, 404 is resource not found, 500 is an internal server error.
- "The app went down" has specific technical causes: a downed server, unresolved DNS, an expired certificate, a code error. Knowing them lets you narrow down the problem.

What comes next

Chapter 3 gets into the two actors of the conversation you just learned about: the client and the server. You already know how information travels between them. What's next is understanding what each one does in that exchange — and why that division is the fundamental structure of any application.

This sample ends here.

The complete book continues with the full map:
client, server, databases, code,
APIs, security, artificial intelligence, and deploy.

The complete edition includes
22 chapters, practical appendices,
a technical glossary, and PDF + EPUB versions.