

Unlock Your Potential and Master Web Development

PROGRAMMING STEP BY STEP AND MORE

AN INSTRUCTIVE GUIDE

BOOK 6

**Full-Stack Engineering with JavaScript:
From Modern Frontend to Modular
Architecture and Cloud Deployment.**

**THE LEAP TO THE BACKEND
THE ENGINEERING YOU DON'T SEE**

SECOND MODULE:

**Modern Backend with JavaScript: Data
Persistence, REST APIs, Modular
Architecture, and Cloud Deployment.**

René F. Ruano Domínguez

Copyright © 2026 René F. Ruano Domínguez

All rights reserved.

DEDICATION

TO FUTURE GENERATIONS, MILLENNIALS, BABY BOOMERS, AND BEYOND.

This guide is dedicated to everyone: To you, the brilliant young minds shaping the world with your ingenuity and creativity. To millennials, who navigate the digital age with fluency and constantly seek new ways to innovate. To baby boomers, who pioneered the technological revolution and now contribute their experience and wisdom. To retirees who dream of staying up-to-date, spending their free time productively, and keeping their minds active.

PROLOGUE

This Sixth Book, titled **"Full-Stack Engineering with JavaScript: From Modern Frontend to Modular Architecture and Cloud Deployment"**, marks a turning point in the "Programming Step-by-Step and More" series. It represents not only the culmination of a progressive learning journey but also a significant contribution to contemporary software development and an up-to-date understanding of the technologies underpinning modern programming. This volume marks the transition from visible frontend code to the deep, underlying logic of the backend, expanding language learning into a comprehensive view of software engineering as applied to real-world systems and professional environments.

Given its broad scope and highly specialized content, Book 6 is presented in two complementary modules that structure the learning process in a clear, progressive manner:

a) The first module, **"From Frontend Development to Modern Service-Based Architecture"**, guides the reader from building dynamic interfaces to understanding the principles underpinning distributed applications and API-based systems.

b) **This Second Module, "Modern Backend with JavaScript: Data Persistence, REST APIs, Modular Architecture, and Cloud Deployment"**, delves into server-side engineering, covering everything from professional data management to the construction of scalable architectures and their deployment on state-of-the-art cloud platforms.

More than just a technical volume, this book is the result of a continuous journey of knowledge, practice, and critical reflection regarding the construction of modern applications. Behind these pages lie years of study, experimentation, and systematic work with JavaScript, coupled with the conviction that knowledge attains its greatest value when shared with clarity, rigor, and a pedagogical approach. This philosophy, which runs through the entire series, is solidified in this sixth book as a mature expression of the author's commitment to teaching and the structured transmission of technical expertise.

This volume reaffirms an educational approach that transcends the mere presentation of tools. Its purpose is to cultivate professional judgment: the ability to analyze problems, design coherent solutions, structure scalable systems, and make responsible decisions within real-world development contexts. Rather than simply teaching how to program, it seeks to teach how to think using engineering logic.

In this regard, the book guides the reader through the transition toward a software engineering mindset—one that is critical, methodical, and creative, and committed to quality, security, and ethics in technological development. Each chapter is designed to reinforce this perspective by integrating practice, analysis, and technical reflection.

Driven by this conviction, we present **Book 6 of the Programming Step by Step and More series**. This work marks the transition from writing code to building systems, transforming accumulated knowledge into solid professional expertise and preparing the reader for the challenges of modern development in the era of Full Stack and cloud computing.

With gratitude and admiration for the author:

Idalmy Baluja Conde

CONTENT

ID	TITLE	PAG
1	DEDICATION	iii
2	PROLOGUE	v
3	ACKNOWLEDGEMENTS	xiii
4	OTHER BOOKS BY THE AUTHOR	xiv
5	BOOK 6 — FULL-STACK ENGINEERING WITH JAVASCRIPT	1
6	INTRODUCTION – SECOND MODULE	3
7	DATABASES AND DATA PERSISTENCE	5
8	Data Persistence	5
9	Local JSON	6
10	What is JSON Local?	7
11	Advantages of using local JSON	8
12	Limitations of Local JSON	8
13	JSON and JavaScript	8
14	When to use local JSON	9
15	Files	9
16	Files as a persistence mechanism	10
17	Types of files used to store data	10
18	Advantages of file-based storage	11
19	Disadvantages of file-based storage	11
20	File handling in Node.js	12
21	When should files be used?	12
22	Demonstration project: JSONLocal + File	13
23	Project architecture	15
24	Scripts for backend files	19
25	Analysis of frontend files	26
26	Publishing the data-persistence project	43
27	SQL	44

ID	TITLE	PAG
28	What is SQL?	45
29	LMS menu project	45
30	CRUD Operations in SQL	46
31	INSERT — Create records	46
32	SELECT — Query information	48
33	UPDATE — Updating records	49
34	DELETE — Deleting records	50
35	CRUD and SQL in Web Applications	50
36	SQL and Enterprise Persistence	51
37	MySQL and phpMyAdmin	53
38	NoSQL	54
39	Differences between SQL and NoSQL	55
40	Document Databases	57
41	Types of NoSQL Databases	57
42	Advantages of NoSQL	59
43	When to Use NoSQL?	59
44	SQL vs NoSQL in Modern Projects	59
45	Hybrid Architecture	60
46	Conclusions for Chapter 1	61
47	MODERN BACKEND DEVELOPMENT ENVIRONMENT	63
48	Why PHP Is No Longer Enough	63
49	Evolution Toward Node.js	64
50	JavaScript Runtime	66
51	The Terminal	66
52	npm	67
53	package.json	68
54	VS Code — Installing Dependencies	69
55	EXPRESS.JS AND API BUILDING	71

ID	TITLE	PAG
56	Comparing Node.js and Express	71
57	Facility	73
58	Server startup	78
59	Checking routes	79
60	Middleware	80
61	Execution Middleware	82
62	Logging Middleware	82
63	Authentication Middleware	83
64	Adding middleware to classroom-messages	84
65	Modifications to project structure	84
66	Verification of new structure	89
67	GET Method	91
68	POST Method	92
69	Integrating POST method	93
70	Verifying POST functionality	96
71	Installing Postman	97
72	Verification process after Postman	98
73	JSON	99
74	REST APIs	100
75	REST API operations in classroom-messages	101
76	Server results after refactoring	107
77	Steps to create a new record	109
78	Steps to update a record	110
79	Procedure for deleting a record	113
80	CONNECTING APIs TO DATA SOURCES	116
81	Memory as a Temporary Data Source	116
82	Memory in Node.js	117
83	In-Memory Task Manager Project	118

ID	TITLE	PAG
84	Minimal project structure	119
85	Verification of full API flow	123
86	Tests performed	134
87	Data in JSON files	136
88	From memory to persistent storage	136
89	Comparison: memory vs JSON files	153
90	Separation of API and persistence	153
91	Changing data source without changing API	155
92	Preparing project for PostgreSQL	157
93	Project connected to PostgreSQL	163
94	Connecting to PostgreSQL locally	164
95	Verifying DB connection	168
96	Modifications for PostgreSQL	169
97	Installing PostgreSQL via Docker	185
98	Docker architecture	185
99	Modifications for Docker storage	191
100	Connecting to PostgreSQL in the cloud	195
101	Advantages of cloud Postgres	196
102	What is a PostgreSQL cloud service?	197
103	Leading PostgreSQL Cloud Providers	200
104	Project architecture	201
105	Git repository as deployment engine	204
106	Steps to build local repository	205
107	Vercel automated process	205
108	package.json file	206
109	.gitignore file	207
110	Workflow: Local → Remote	209
111	Lifecycle: Local → Remote	212

ID	TITLE	PAG
112	Files of memory-task-manager	214
113	PostgreSQL connection file	215
114	API Engine (server.js)	220
115	Client Module (main.js)	238
116	User Interface (index.html)	247
117	Local testing before deployment	249
118	Adjustments for local testing	250
119	Endpoint Validation Matrix	255
120	Deploying to Vercel	259
121	Review of architecture	260
122	Adjustments to vercel.json	261
123	Updating local repository	264
124	Operations for Vercel deployment	266
125	Connecting remote project to Vercel	266
126	Conclusions of Chapter 4	281
127	MODULAR ARCHITECTURE AND DESIGN PATTERNS	283
128	Modularizing the API	283
129	Separation of Concerns	285
130	MVC pattern applied to Express	288
131	The Service Layer	292
132	Repository Pattern	295
133	Professional code organization	296
134	SECURING AND PROFESSIONALIZING JS APPLICATIONS	302
135	General execution flow	302
136	JavaScript code minification	304
137	JavaScript code obfuscation	305
138	Obfuscation — Medium complexity	315

ID	TITLE	PAG
139	Obfuscation — High complexity	319
140	Obfuscation — Higher minification	324
141	Chapter conclusions	329
142	GENERAL CONCLUSIONS	331
143	REFERENCES	333
144	MOTIVATIONAL PHRASES	337
145	ABOUT THE AUTHOR	339

Si

ACKNOWLEDGEMENTS

To my family, my colleagues, and my friends.

Thank you—truly, a thousand times—for the unconditional support you have given me since I first embarked on this project.

And to those who did not believe in it: thank you as well. Your doubts became motivation and challenged me to prove that the effort was worthwhile.

Without your encouragement, suggestions, trust, and support—and without the personal challenge of creating something meaningful and useful—this project would not have been possible.

.

OTHER BOOKS BY THE AUTHOR

If you wish to continue exploring the "Programming Step-by-Step and More" collection, you can access all volumes, updated editions, and new releases available on Amazon via the QR code below.

The entire series is published on Amazon.

Scan the QR code to discover more titles, supplementary resources, and upcoming releases.



BOOK 6

**Full-Stack Engineering with JavaScript:
From Modern Frontend to Modular
Architecture and Cloud Deployment.**

Second Module:

**Modern Backend with JavaScript: Data
Persistence, REST APIs, Modular
Architecture, and Cloud Deployment.**

René F. Ruano Domínguez

I- BOOK 6 — FULL-STACK ENGINEERING WITH JAVASCRIPT: FROM MODERN FRONTEND TO PROFESSIONAL BACKEND AND SYSTEM SECURITY.

Book 6 comprises a training package made up of three complementary modules, designed to guide the student from modern frontend development to professional service-based architecture, JavaScript-based backend development, and quality and security engineering. Each module is published as a standalone edition to balance content volume, reduce costs for the student-reader, and maximize the practical value added to their professional profile and quality of life.

The First Module, "From Frontend Development to Modern Service-Based Architecture", introduces the reader to the fundamentals required to evolve from browser-based code to distributed systems. Across six sections, students work with JSON, templates, asynchrony, Promises, and client-server communication, ultimately building the foundation of a multi-user SaaS system—a controlled test case—by integrating modularity, data handling, and remote services.

This Second Module, "Modern Backend with JavaScript: Data Persistence, REST APIs, Modular Architecture, and Cloud Deployment" covers the fundamentals of modern backend development through a highly practical approach. It begins with the principles of data persistence, examining in-memory storage, JSON files, and SQL and NoSQL databases; proceeds to explore the Node.js ecosystem, npm, and Express.js for building servers and REST APIs; moves on to integrating PostgreSQL as a data source using CRUD operations; and concludes with professional software organization—employing modular architecture, the MVC pattern, and service and repository layers—as well as cloud application deployment via Git, GitHub, and Vercel, and the optimization and protection of JavaScript code through minification and obfuscation techniques.

Each module follows a methodology in which students do not simply study theoretical concepts but immediately apply them through practical development. Throughout the learning process, they complete hundreds of exercises and progressively build real-world projects by writing every line of code in Visual Studio Code, creating project structures, executing scripts, and monitoring application behavior in

real time. This hands-on approach transforms learning into an active experience in which students observe how their functions, classes, APIs, and modules are rendered in the browser, analyze their execution using developer tools, and validate backend functionality through a local development server. The result is technical training that closely mirrors the workflow of a professional software development environment.

Furthermore, each module is further complemented by practical examples published on the Energyner learning platform (<https://energyner.net/jscript/>), where students can compare their own solutions, validate results, and explore alternative implementations. Together, the books, the online platform, the development environment, browser developer tools, local servers, version control, and cloud deployment form a comprehensive learning ecosystem that guides students through the complete software development lifecycle—from writing source code to building APIs, managing databases, deploying cloud applications, and organizing professional software architecture. Rather than focusing on isolated technologies, this ecosystem helps students develop the analytical thinking and engineering practices required for modern Full Stack software development.

Upon completing the two modules that comprise Book 6, readers will have developed a comprehensive understanding of modern Full Stack software engineering with JavaScript. They will understand how frontend applications communicate with backend services, how APIs integrate with databases, how software is organized using professional architectural patterns, how applications are deployed to cloud platforms, and how code is optimized and protected before production release. Equipped with both practical experience and architectural insight, they will be prepared to design, build, deploy, and maintain real-world applications using the technologies, workflows, and engineering practices adopted throughout today's software industry.

II- INTRODUCTION – SECOND MODULE.

During the early years of web development, much of the focus was on user interface design and browser interaction. However, behind every modern application lies an infrastructure responsible for processing information, managing data, responding to user requests, and coordinating communication between multiple services. This infrastructure constitutes the backend—the true engine that powers today's applications.

In this second module of Book 6, we will transition from client-side development to the world of modern backend development, using JavaScript as our primary language and leveraging the tools and technologies that currently form one of the software industry's most widely used development ecosystems.

Our journey begins with the fundamentals of data persistence, analyzing how applications store information using memory, JSON files, and databases. We will then introduce relational models via SQL and PostgreSQL, as well as the core concepts of NoSQL databases and the hybrid architecture used in enterprise applications.

Next, we will explore the professional Node.js development environment, gaining an understanding of the JavaScript runtime outside the browser, the use of the terminal, the npm package manager, and the tools that enable the efficient and organized construction of modern backend applications.

Building on this foundation, we will introduce Express.js—the most popular framework in the Node.js ecosystem—learning how to develop HTTP servers, implement middleware, build REST APIs, and establish standardized communication mechanisms between client applications, services, and databases.

As the module's practical core, we will develop the "memory-task-manager" project step-by-step. Its architecture will evolve progressively: starting as an application that stores data solely in memory, it will transform into a professional solution connected to a PostgreSQL database, ready to run in local environments and via Docker containers, and finally deployed to the cloud using Git, GitHub, and Vercel. The module also delves into internal software organization through modular

architecture, the separation of concerns, and the application of design patterns widely used in professional development—such as Model-View-Controller (MVC), Service Layer, and Repository Pattern—demonstrating how these techniques enhance an application's maintainability, reusability, and scalability.

Finally, we will examine how to prepare an application for production deployment using JavaScript code optimization and protection techniques, incorporating minification and obfuscation processes with industry-standard tools like Terser and JavaScript Obfuscator.

The structure of this second module naturally addresses the questions that arise during the evolution of any modern application:

How can data be stored persistently?

How do you build a server using JavaScript?

How do you design and implement a REST API?

How do you connect an application to a relational database?

How do you organize code using a professional architecture?

How do you deploy an application to the cloud?

How do you optimize and protect code before publication?

By the end of this module, the reader will have navigated the entire lifecycle of building a modern backend application: from data persistence and API development to software architecture, cloud deployment, and JavaScript code protection. Rather than simply learning isolated tools, the reader will gain a comprehensive view of the professional development process, establishing a solid foundation for tackling more complex projects and the technologies that shape today's web development ecosystem.

1. DATABASES AND DATA PERSISTENCE.

Before building servers, APIs, or full-scale applications, it is essential to understand where and how information is stored. Every modern application needs to retain data beyond a program's execution, ensuring that user information, configurations, logs, and operational data remain available after the system restarts. This process is known as data persistence and is a fundamental pillar of backend development.

In this chapter, we will explore various methods of storing information, starting with simple solutions like local files and JSON documents before delving into the world of databases. We will examine the core concepts of relational systems, the SQL language, and the use of MySQL—one of the industry's most widely used database management systems.

Finally, we will look at the NoSQL approach and the scenarios where this type of database offers advantages over traditional models.

By the end of this chapter, the reader will understand key data storage strategies and be prepared to implement persistence mechanisms in modern applications built with JavaScript and current backend technologies.

1.1. Data Persistence

One of the most important aspects of software development is the ability to preserve information over time. When an application runs, the data it uses is stored temporarily in the system's memory. However, when the program is closed or the device is turned off, that information is lost. To prevent this data loss, data persistence mechanisms are used.

Data persistence is the process by which information generated or used by an application is stored on a permanent medium, allowing it to remain available even after the program has finished executing. Thanks to this principle, it is possible to save user accounts, passwords, settings, activity logs, products, orders, and any other information necessary for a system's operation.

Without data persistence, every time an application started, it would begin from scratch, losing all history and previously stored information.

For this reason, persistence is a fundamental pillar of any modern application.

There are various mechanisms for achieving data persistence. The most common include:

- Text files.
- JSON files.
- XML files.
- Relational databases.
- NoSQL databases.
- Cloud storage services.

The choice of a persistence mechanism depends on factors such as the volume of information, query complexity, required performance, and application scalability.

For example, a small application might store information in local JSON files, whereas an e-commerce platform with thousands of users would require a database capable of managing large amounts of data efficiently and securely.

As applications grow, so do the needs for data storage, organization, and retrieval. Therefore, understanding the fundamentals of persistence is essential before studying specific technologies like SQL, MySQL, or NoSQL databases.

In the following sections, we will explore different ways of storing information, starting with simple file-based methods and progressing to database systems used in professional environments.

1.1.1. Local JSON

One of the simplest ways to implement data persistence is by storing information in JSON (JavaScript Object Notation) files. This format has become a widely used standard due to its simplicity, readability, and compatibility with multiple programming languages. (See Chapter 1: JSON - Data Processing – Book 6 – First Module)

JSON allows data to be represented using object and array structures based on key-value pair syntax. Thanks to its text-based format, JSON files can be read by both humans and computer programs.

An example of a JSON file is as follows:

Cuadro 1. JSON file

```
{
  "name": "Juan Pérez",
  "age": 28,
  "active": true
}
```

In this case, the file stores information about a user using a set of properties organized within an object.

1.1.2. What is JSON Local?

When we talk about local JSON, we mean storing data in JSON files located within an application's file system. Instead of using a database, the information is saved directly in one or more files that can be read and modified by the program when necessary.

For example, an application may have a file called:
users.json
with the following content:

Table 1. Users.json file

```
[
  {
    "id": 1,
    "name": "Ana"
  },
  {
    "id": 2,
    "name": "Carlos"
  }
]
```

Whenever the application needs to query the users, it will simply read this file and convert its content into JavaScript objects.

1.1.3. Advantages of using local JSON

Storing data in JSON files offers several advantages for small projects and learning environments:

- Easy to implement.
- Does not require installing a database server.
- Human-readable and easy-to-edit format.
- Natively compatible with JavaScript.
- Ideal for prototypes and simple applications.

For these reasons, JSON is often the first persistence solution developers use when starting out with Node.js.

1.1.4. Limitations of Local JSON

While practical for small projects, JSON has significant limitations as the amount of data grows:

- Searches can become slow.
- There is no advanced query system.
- It does not handle complex data relationships.
- Conflicts can arise when multiple users attempt to modify the same file simultaneously.
- It is not designed for large volumes of data.

Due to these limitations, professional applications often migrate to specialized database systems when they require greater performance, security, and scalability.

1.1.5. JSON and JavaScript

One of the main reasons for JSON's popularity is its close relationship with JavaScript. Node.js includes tools that make it easy to convert JavaScript objects to JSON format and vice versa.

For example:

Table 1. Conversion of JS objects to JSON format and vice versa.

```
#=====Object JS to JSON=====  
const user= {  
  name: "Ana",  
  age: 25  
};  
const json = JSON.stringify(user);
```

```
console.log(json);  
#Result:  
{ "name": "Ana", "age": 25 }  
#=====JSON to JS format=====  
const object= JSON.parse(json);  
console.log(object.name);  
Result:  
Ana
```

These functions greatly facilitate the storage and retrieval of information within applications developed with JavaScript.

1.1.6. When to use local JSON

Using JSON files is recommended in situations such as:

- Educational projects.
- Small personal applications.
- Rapid prototypes.
- Configuration files.
- Low-complexity temporary data.

However, when an application needs to handle large amounts of information or multiple simultaneous users, it is more appropriate to use a specialized database.

Local JSON serves as an excellent starting point for understanding data persistence concepts and acts as a natural bridge to the study of more advanced storage systems, which we will cover in the following sections.

1.2. Files

Before the widespread adoption of modern databases, many applications stored their information directly in files within the operating system. In fact, even today, numerous programs continue to use files to save configurations, logs, documents, and other types of information.

A file is a storage unit that allows data to be saved permanently on a physical device—such as a hard drive, a solid-state drive (SSD), or a cloud storage system. Files constitute one of the most fundamental

mechanisms for data persistence and serve as the foundation upon which many more advanced storage technologies operate.

1.2.1. Files as a persistence mechanism

When an application saves information to a file, the data remains stored even after closing the program or turning off the computer. Subsequently, the application can reopen the file and retrieve the previously saved information.

For example, an application might store a list of users in a text file: `users.txt`.

- Juan Pérez
- Ana García
- Carlos López
- María Fernández

Whenever the program needs to query the registered users, it will simply read the contents of the file.

This approach is simple and effective for small applications, although it has limitations when the amount of information grows or when complex searches are required.

1.2.2. Types of files used to store data

There are various file formats that can be used to store information:

a) Plain text files

These are the simplest and contain only human-readable characters.

Example:

- Name: Juan
- Age: 28
- City: Madrid

Their main advantage is ease of reading, although they do not offer an organized structure for large volumes of information.

b) CSV files *CSV (Comma-Separated Values)*

It is a widely used format for storing tabular data.

Example:

id,name,age

- 1,Ana,25
- 2,Carlos,30
- 3,María,27

CSV files are compatible with spreadsheets and data analysis systems.

c) JSON Files

They allow for the storage of structured information using objects and arrays. (We saw this earlier)

Example:

```
{
  "name": "Ana",
  "age": 25
}
```

Due to its integration with JavaScript, JSON has become one of the most widely used formats in modern web applications.

d) XML files

XML (*eXtensible Markup Language*) for years, it was one of the standards for exchanging information between systems.

Example:

- <user>
- <name>Ana</name>
- <age>25</age>
- </user>

This format is still used in some enterprise environments, although JSON is currently often the preferred choice due to its simplicity.

1.2.3. Advantages of file-based storage

Using files offers several benefits:

- Easy implementation.
- Does not require installing a database management system.
- Low resource consumption.
- Ideal for small applications.
- Allows for storing configurations and simple data.

For this reason, many programs use files to save user preferences, activity logs, or internal configurations.

1.2.4. Disadvantages of file-based storage

As an application grows, various limitations arise:

- Searches can be slow.
- There are no advanced query mechanisms.
- Maintaining relationships between different datasets is difficult.
- Concurrency can lead to conflicts when multiple processes access the same file simultaneously.
- Maintenance becomes complex as the volume of information increases.

These limitations drove the development of specialized systems for data management: databases.

1.2.5. File handling in Node.js

Node.js provides a native module called ``fs`` (File System) that allows for interaction with files and directories. This gives us the ability to read, write, modify, rename, and delete files and folders directly from JavaScript code. For example, to read a file:

Box 1. Node.js native module – ``fs``

```
//Read a file  
import fs from "fs";  
const contenido = fs.readFileSync("usuarios.txt", "utf-8");  
console.log(contenido);  
  
//To write information  
import fs from "fs";  
fs.writeFileSync("mensaje.txt", "Hello from Node.js");
```

These operations make it possible to create basic persistence systems without the need for a database.

1.2.6. When should files be used?

File-based storage is suitable for:
Application configurations.

- Log files.
- Temporary data.
- Educational projects.
- Small prototypes.

- Data export and import.

However, when an application needs to manage large volumes of data, perform complex queries, or support multiple simultaneous users, using a database is recommended.

Files represent the most basic level of data persistence and serve as an excellent starting point for understanding how applications store information. Modern databases emerged to address these needs, designed to provide greater organization, performance, security, and scalability.

1.3. Demonstration project regarding data persistence in JSONLocal and a file.

We will carry out this exercise to gain the skills needed to integrate these two essential data persistence mechanisms into future projects. These techniques become go-to solutions when building systems that—by their very nature—require transferring information between pages or maintaining state throughout a process (such as in sequential calculations, authentication flows, multi-step forms, and other scenarios where data continuity is essential).

The sequence we will program follows this pattern:

form --> app.js --> fetch() / Node.js --> usuarios.json --> Hard Drive

The browser CANNOT directly write arbitrary files to the user's hard drive. Therefore, we will need:

Frontend --> Backend with Node.js

a) Development environment

Since this project combines static components (HTML, CSS, and the interface, which handle data storage in JSONLocal) with dynamic components (an Express backend responsible for reading and writing data on the server), the backend save process requires an environment capable of handling both elements in an integrated manner. This entails having a system that can serve static files while simultaneously executing the Node.js and Express code that manages data persistence.

In the local environment, we will use two complementary tools: Visual Studio Code and Wampserver. Wampserver provides a fully functional Apache server, allowing us to deploy the project locally and navigate it just as if it were hosted on a live server.

At the same time, the integrated terminal in Visual Studio Code allows us to install all necessary project dependencies, manage Node.js modules, and run the Express server in development mode.

Combining these platforms enables us to replicate a complete working environment where Apache serves the project's static component, while Node.js and Express execute the dynamic component responsible for data persistence.

This allows us to interact with the project under conditions very similar to its online deployment, validate its functionality, and debug any errors before moving it to a production environment.

b) Production environment.

In a production environment, a project containing both static and dynamic components requires execution on a platform running an Express server. Without an active backend, dynamic persistence operations—such as saving users, reading the `users.json` file required by the system, or deleting records—cannot be performed, as static hosting services do not provide file system access or support for Node processes.

Traditional shared hosting providers—such as Bluehost, HostGator, iPage, or GoDaddy—do not offer an execution terminal or an environment capable of running Node.js. These hosting services are limited to static sites and PHP applications; consequently, they cannot execute a dynamic backend or launch an Express server.

We currently use a shared server to publish the projects developed throughout this guide; therefore, the dynamic component will remain completely non-functional in that setup.

To run both the static and dynamic components, a VPS or dedicated server is required—where it is possible to install Node.js, configure Express, and enable data persistence. Alternatively, one can use a cloud

server platform—such as Vercel, Render, Google Cloud, or Railway—which offers all the capabilities needed to run a complete, professional application (both static and dynamic), including the backend, API routes, and file handling.

These platforms—whether physical servers or cloud-based—will be examined in depth in the upcoming Books 7, 8, and 9, where we will learn how to deploy dynamic projects in professional, scalable environments.

1.3.1. Project architecture

We will work with a simple yet highly representative architecture of how modern web projects is organized. The goal is for the student to understand, from the outset, the separation between the project's public section—where frontend resources reside—and the private or internal section—where server logic and data persistence are managed. The project structure is divided into two main areas:

assets/: contains all frontend resources (documentation, styles, scripts, and visual components).

server/: houses the backend logic, including the Node.js server and data files that simulate a real database.

data/: serves as the server's internal storage. Files acting as the application's persistent data source are kept here. In this project, the ``usuarios.json`` file acts as a small local database where the backend records, queries, and updates information sent from the frontend.

This organization enables the development of modern, easily scalable, and maintainable applications, while also introducing the student to a professional workflow—one they will encounter in real-world projects and which we will continue to develop later.

We are creating the ``15-advanced-backend`` folder, a name that accurately reflects the content covered in this second module of Book 6. This folder maintains the same organizational logic used in the core project developed throughout the Programming Step-by-Step and More guide and integrates seamlessly with the Learning Management System structure. In this way, we ensure continuity, order, and consistent architecture across all the book's modules.

Within this folder, we will create the different domains for each practical project, developing them progressively to practice concepts, reinforce skills, and build complete solutions based on the topics covered in this second module of Book 6.

We have named this first project `data-persistence`.

Table 1. Architecture of the data-persistence project

```

└─ data-persistence
  └─ assets
    └─ app
      └─ JS template-app.js
    └─ css
      └─ # style.css
    └─ doc
      └─ < index-article.html
    └─ js
      └─ JS app.js
  └─ data
    └─ {} usuarios.json
  └─ server
    └─ JS server.js
    └─ < index.html
```

We will describe the files and their functions:

root: data persistence

It contains all the specific resources necessary to execute the data-persistence educational module, maintaining a clear separation between the global files that are part of the Learning Management System (LMS) template and the particular files of the project. This organization allows the practice to be integrated within the LMS without mixing common components with the customized elements that belong exclusively to this module.

root/index.html

It is the entry point of the project. Its main function is to load the global LMS template and register the project's own resources. It acts as a bridge between the overall LMS infrastructure and the custom components of the project.

root/assets/app/template.js

It is the assembly engine of the project. Your responsibility consists of:

- Create the modular structure of the page.
- Register the HTML components that need to be loaded.
- Locate the content files.
- Dynamically inject modules into the DOM.
- Coordinate the rendering process.
- Notify when all components have been loaded.

In other words, it functions as the chief engineer who builds the final interface that the user will see.

root/assets/css/template.js

Contains the visual styles unique to the data persistence project.

root/assets/doc/index-article.html

It contains the academic content and the interface of the practical exercise.

Store:

- Theoretical explanations.
- Instructions for the student.
- Data capture forms.
- Containers where the results will be displayed.

During execution, the file is dynamically loaded by the template system and subsequently inserted into the main document.

root/assets/js/app.js

It represents the functional core of the project. Your responsibility consists of:

- Capture form events.
- Validate information entered.
- Create JavaScript objects.
- Convert data to JSON format.
- Manage information persistence.
- Communicate with the server when necessary.
- Retrieve stored records.
- Update the UI when the DOM is finished.

It is the file where the business logic of the exercise resides.

root/server/server.js

Implement the backend server of the project using Node.js and Express. Its main functions are:

- Receive HTTP requests.
- Process requests sent from the browser.
- Read files stored on disk.
- Write new records.
- Manage CRUD operations.
- Expose REST APIs for the frontend.

This component allows you to evolve the project from local persistence based on localStorage to real persistence using files.

root/data/users.json

It is the physical repository of project data. It acts as a small file-based database. Its function is to permanently store the information registered by users, including:

- Identifier.
 - Name.
 - Email.
 - Other fields that may be incorporated later.

Unlike localStorage, this file remains stored directly on the disk of the computer where the Node.js server is running, ensuring information persistence even after closing the browser or restarting the application.

We have, therefore, reached the point where it is necessary to launch the backend so that the functions responsible for saving and managing data can run correctly on the server.

The process is carried out in two branches or directions.

The form sends the data to the app.js file, which:

Branch 1. Local JSON

- Saves the information to localStorage.
- Updates the Local JSON Panel, which represents the user view.

This branch runs without a server, entirely within the browser.

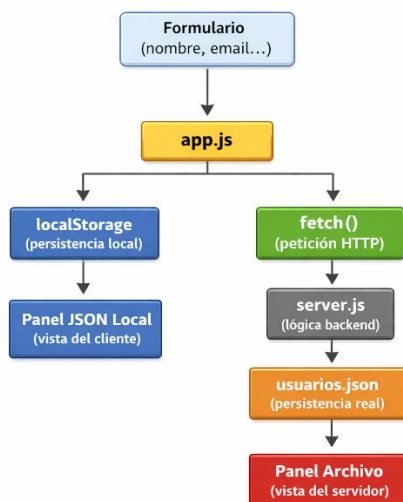
Branch 2. Server-side persistence (backend)

Simultaneously, app.js:

- Send the data using `fetch()`.
- `server.js` receives the request.
- Saves the data to `usuarios.json`.
- The File Panel displays the actual content of the file on disk.

This branch represents actual persistence, outside the browser.

Table 2. Process flow diagram



We will create the ``package.json`` file, which serves a dual purpose:

- Defining the project's identity.
- Managing the dependencies required to run the Node.js server responsible for file-based persistence.

The goal of this chapter is to teach data persistence using local JSON and files, so it is best to keep the environment simple and easy to understand.

1.3.2. Script for the project's main backend files

a)package.json

As we recall from Book 5 of the series—where we first implemented the code for this file and analyzed its internal structure—the ``package.json`` file serves as the configuration core for any Node.js-

based project. It defines project metadata, dependencies, execution scripts, and environment versions, acting as the central hub that coordinates the system's installation, execution, and maintenance.

Table 1. Script and structure of the project's package.json file

<pre>{ "name": "data-persistence", "version": "1.0.0", "description": "Project: Data persistence using local JSON and files, utilizing Node.js and Express.", "type": "module", "main": "server/server.js", "scripts": { "start": "node server/server.js", "dev": "node server/server.js" }, "keywords": ["nodejs", "express", "json", "localstorage", "persistencia", "files"], "author": "EnergyNer LMS", "license": "MIT", "dependencies": { "cors": "^2.8.6", "express": "^5.2.1" }, "devDependencies": {} }</pre>
--

Next, we will review the files and scripts that make up the backend—the reserved area of the project where server-side logic executes and actual data persistence is managed.

The steps we have implemented allow for a clear, organized view of how the algorithm creates, reads, updates, displays, and—when necessary—deletes records. This walkthrough makes it easier to

understand the full CRUD operation cycle and how each action is reflected on both the client and the server.

Server file server.js

The server.js file is a basic backend server built with Node.js and Express; its primary function is to act as a REST API that manages a data persistence system using local files.

Instead of using a formal database (such as PostgreSQL or MySQL), it uses a JSON-formatted text file (users.json) to store, read, and delete user information.

Table 2. server.js script

```
// server.js

import express from "express";
import cors from "cors";
import fs from "fs";
import path from "path";

const app = express();
const PORT = 3000;

const DATA_FILE =
path.resolve(
  "./data/usuarios.json"
);
// =====
// MIDDLEWARE
// =====
app.use(cors());
app.use(express.json());
console.log("Initializing server...");

// =====
// VERIFY DATA FILE
// =====
if (!fs.existsSync(DATA_FILE)) {
  console.log(" usuarios.json does not exist. Creating file....");
```

```

fs.writeFileSync( DATA_FILE, "[]");

}
// =====
// RUTA RAÍZ
// =====
app.get("/", (req, res) => {

  console.log( " Application received on/");

  res.send( "Data Persistence server functioning correctly..");

});

// =====
// GET USERS
// =====
app.get("/usuarios", (req, res) => {

  console.log( "GET /usuarios");

  try {
    const usuarios = JSON.parse(
      fs.readFileSync( DATA_FILE, "utf8" ) );

    console.log( ` Users found: ${usuarios.length}` );
    res.json(usuarios);

  } catch(error) {
    console.error(
      "✗ Error reading usuarios.json",
      error
    );

    res.status(500).json({
      ok: false,
      mensaje:
        "Error reading user file"
    });
  }
};

```

```

}

});
// =====
// SAVE USER
// =====
app.post("/usuarios", (req, res) => {

  console.log( "POST /users received");

  try {
    const usuario = req.body;

    console.log("users received:", usuario );

    const usuarios = JSON.parse(
      fs.readFileSync(
        DATA_FILE,
        "utf8"
      )
    );

    usuarios.push(usuario);

    fs.writeFileSync(
      DATA_FILE,
      JSON.stringify( usuarios, null, 2 )
    );

    console.log( "User saved successfully." );

    res.json({ ok: true, mensaje: "User stored",
      total:
        usuarios.length
    });

  } catch(error)
  {
    console.error(
      "✗ TOTAL FAILUREPOST /usuarios"
    );
  }
}

```

```

    );
    console.error(error);
    res.status(500).json({
ok:false,
    mensaje:
        "Error saving user",
    detalle:
error.message
    });
}

});

/**=====
 * DELETE/USERS
 * ===== */
app.delete("/usuarios",
(req,res)=>{

    console.log( " DELETE /users" );
    try {
        fs.writeFileSync(
            DATA_FILE,
            JSON.stringify(
                [],
                null,
                2
            )
        );
        console.log( " File restarted" );

res.json({
    ok:true,
    mensaje:
"File cleaned"
    });
    } catch(error){
        console.error(
"✗ Error deleting users",
        error

```

```

    );

    res.status(500).json({

        ok:false,

        mensaje:
        "Error deleting users"
    });
}
});

// =====
// SERVER STARTUP
// =====

app.listen(PORT, () => {

    console.log( `Server running in:    http://localhost:${PORT}`
    );

});

```

Key server functions

a). Initial configuration

- Imports Express, CORS, and the native `fs` and `path` modules.
- Defines the port (3000) and the absolute path to the data file.
- Checks if `usuarios.json` exists; if not, it automatically creates it with `[]` to prevent errors during the first run.

b). Essential middleware

- `cors()` enables requests from other origins (useful for frontends running on different ports).
- `express.json()` allows the server to receive and process JSON data from the client.

c). API endpoints (basic CRUD)

- GET / → Simple response to verify the server is running.
- GET /usuarios → Reads and returns the contents of the JSON file.
- POST /usuarios → Receives a user, adds them to the array, and overwrites the file.

- DELETE /usuarios → Completely clears the file, removing all records.

d). Error handling and logging

- Each operation is wrapped in a `try...catch` block to prevent server crashes.
- In the event of an error, it responds with an **HTTP 500** status and logs details to the console.
- Includes descriptive console messages to monitor the data flow.

User data record file: users.json

The `usuarios.json` file serves as the backend's persistent data store. Its purpose is to store—in a simple, structured manner—the information managed by the server through CRUD operations. Although it is not a formal database, it acts as a local repository that allows for the simulation of real persistence systems within an educational setting.

Its structure is minimal: the file begins with an empty array `[]`, indicating that it will store a collection of JSON objects. As the server receives new records, they are added to the array, turning the file into a simple, human-readable representation of the current data state.

1.3.3. Analysis of the project's frontend files

Next, we will analyze the main frontend files—that is, the part of the application where the user enters data and where scripts execute to manipulate that information, send it to the server, receive backend responses, and display results in the browser. We will also examine the mechanism that, upon clicking the "Clear Data" button, deletes records stored in both the local view and the server file.

File handling the logic within the workflow

Table 1. app.js file

```
// app.js
// =====
// ENVIRONMENT CONFIGURATION
// =====
const CONFIG = {
  DEVELOPMENT_API: "http://localhost:3000",
```

```

PRODUCTION_API: "https://energyner.net"
};

const API_URL = window.location.hostname === "localhost"
? CONFIG.DEVELOPMENT_API
: CONFIG.PRODUCTION_API;

console.log( " Current host:", window.location.hostname );

console.log( "Selected API:", API_URL );

// =====
// MAIN APP
// =====

function initApp() {
console.log(
"1- Starting Persistent Contact Manager application..."
);

const form =
document.getElementById("userForm");

const btnClear =
document.getElementById("btnClear" );

const jsonlist =
document.getElementById(
"jsonuserlist"
);

const fileList =
document.getElementById(
"listUsersFile"
);

console.log(
"2- ✓ DOM elements loaded:",
{

```

```

form,
jsonlist,
filelist
}
);

if (
!form ||
!listJson ||
!filelist
) {

console.warn(
"Components not found."
);

return;
}

loadstones'());
loadUsersFile();

form.addEventListener(
"submit",
saveUser
);
btnClear.addEventListener(
"click",
cleanData
);
// =====
// SAVE USER
// =====

async function saveUser(e) {

console.log(
"saveUser-1.1 → Submit event detected"
);

```

```

e.preventDefault();

const name =
document.getElementById(
"name"
).value;

const email =
document.getElementById(
"e-mail"
).value;

console.log(
"1.2- Captured data:",
{
name,
e-mail
}
);

const user = {

id: Date.now(),

name,

e-mail
};

console.log(
"1.3- User created:",
usuario
);

// =====
// LOCAL JSON PERSISTENCE
// =====

const usuariosLocal =
JSON.parse(
localStorage.getItem(

```

```

"usuarios"
)
) || [];

usuariosLocal.push(usuario);

localStorage.setItem(
  "usuarios",
  JSON.stringify(
    usuariosLocal
  )
);

console.log(
  "1.4- User saved to localStorage"
);

// =====
// FILE PERSISTENCE
// =====

try {

  console.log(
    "1.5- Sending user to server..."
  );

  const response =
    await fetch(
      `${API_URL}/usuarios`,
      {
        method: "POST",

        headers: {
          "Content-Type":
            "application/json"
        },

        body:
          JSON.stringify(
            usuario

```

```

)
}
);

console.log(
"1.5.1- Response received:",
response.status
);

const resultado =
await response.json();

console.log(
"1.5.2- Full response - User saved to file:",
resultado
);

} catch(error){

console.error(
" Error saving file:",
error
);
}

form.reset();

console.log(
"1.6- Form reset"
);

cargarUsuariosJson();

cargarUsuariosArchivo();
}

/**=====
* FUNCTION TO CLEAR DATA FROM
LOCALSTORAGE AND THE usuarios.json FILE
* ===== */

```

```

async function limpiarDatos() {

console.log(
"4.1- Starting cleanup..."
);

// =====
// LOCAL STORAGE
// =====

localStorage.removeItem(
"usuarios"
);

console.log(
"4.2- Local JSON deleted"
);
// =====
// FILE
// =====
try {

const response =
await fetch(
`${API_URL}/usuarios`,
{
method:"DELETE"
}
);

const resultado =
await response.json();

console.log(
"4.3- ✓ File cleared:",
resultado
);

} catch(error){

```

```

console.error(
  " Error clearing file:",
  error
);
}

cargarUsuariosJson();

cargarUsuariosArchivo();

console.log(
  "4.4- Interface updated"
);
}
// =====
// LOCAL JSON
// =====

function cargarUsuariosJson() {

  console.log(
    "2.1- Loading Local JSON..."
  );

  const usuarios =
    JSON.parse(
      localStorage.getItem(
        "usuarios"
      )
    ) || [];

  listaJson.innerHTML = "";

  usuarios.forEach(usuario => {

    listaJson.innerHTML += `
    <div class="user">

    <strong>
    ${usuario.nombre}
    </strong>
  `;
  });
}

```

```

<br>

${usuario.email}

</div>
`;
});

console.log(
  "2.2- Local JSON rendered:",
  usuarios.length
);
}
// =====
// usuarios.json FILE
// =====
async function cargarUsuariosArchivo() {

  console.log(
    "3.1- Loading users from file..."
  );

  try {

    console.log(
      "3.1.1- Querying API..."
    );

    const response =
      await fetch(
        `${API_URL}/usuarios`
      );

    console.log(
      "3.1.2- Response received:",
      response.status
    );

    const usuarios =
      await response.json();

```

```

console.log(
  "3.1.3- Users received:",
  usuarios
);

listaArchivo.innerHTML = "";

usuarios.forEach(usuario => {

  listaArchivo.innerHTML += `
<div class="user">

<strong>
${usuario.nombre}
</strong>

<br>

${usuario.email}

</div>
`;
});

console.log(
  "3.2- File rendered:",
  usuarios.length
);

} catch(error){

console.error(
  " Error reading file:",
  error
);
}
}

} // END initApp

```

```
// =====
// TEMPLATE SYSTEM
// =====
/**This listener waits for the Template System to finish:
 * loading HTML modules
 * injecting them into the DOM
 * rendering components
 * completing page assembly
 * When that happens, the Template System (template-app.js
file) fires a custom event called
 * "modulesRendered"
 * When the modulesRendered event is received in this app.js
file, the associated function initApp is executed. * This function
handles data manipulation and processing to output the result
to the browser.
 */

document.addEventListener(

"modulesRendered",

() => {

console.log(
"modulesRendered received"
);
initApp();
}
);
```

The app.js file acts as the main frontend handler, coordinating the entire data flow between the user interface, the browser's local storage, and the backend server. Its function is to initialize the application, capture form data, process it, send it to the server, render the results on screen, and allow for the complete clearing of records.

Key script functions

a) Automatic environment selection (dev / production)

It detects whether the application is running on localhost or a real domain and selects the corresponding API.

This allows the same frontend to work without changes in both environments.

b) Application initialization

The `initApp()` function:

- Gets the DOM elements (form, lists, buttons).
- Load data from `localStorage` and from the server file.
- Connect the events of the form and the Clear Data button.

c) User registration

The `saveUser()` function:

- Capture name and email of the form.
- Creates a user object with a unique ID.
- Save the log to `localStorage`.
- Send the user to the backend using `fetch()` to store it in `users.json`.
- Updates both views: local JSON and server file.

d) Total data cleaning

The `cleanData()` function:

- Delete `localStorage` logs.
- Send a DELETE request to the backend to empty `users.json`.
- Refreshes the interface with both lists updated.

e) Data rendering

`loadJsonUsers()` shows the data saved locally.

`loadUsersFile()` queries the API and displays the actual file data from the server.

f) Integration with the Template System

The script waits for the `modulesRendered` event, emitted by the templating system, to ensure that the DOM is fully assembled before executing `initApp()`.

index-article.html file

The data reaches this handler directly from the user interface form in the ``index-article.html`` file.

The structure and HTML code of this form—including the fields displaying the stored data and the delete button—are shown in the

following box, which illustrates how the fields that will subsequently be processed by the script are captured.

Table 2. Form id: `userForm`

```
<form id="userForm">
<input type="text" id="name" placeholder="Name" required
>

<input type="email" id="email" placeholder="Email"
required >

<button type="submit"> Save </button>
</form>
<div class="persistence-container">
<div class="persistence-panel">
<h2> Local JSON (localStorage) </h2>
<p> Data stored within the browser. </p>
<div id="JsonUserList"></div>
</div><!--Close persistence-panel-->

<div class="persistence-panel">
<h2> Users.json file </h2>
<p> Data physically stored by Node.js on the hard drive. </p>
<div id="listUsersFile"></div>
</div><!--Close persistence-panel-->

<div class="persistence-panel">
<h2> Delete records </h2>
<p> We delete the data stored in the browser's localStorage on
the frontend and in the `users.json` file on the backend. </p>
<button type="button" id="btnClear" > Clear Data
</button>
</div><!--Close persistence-panel-->
```

template-app file

The **`template-app.js`** file is responsible for dynamically assembling the web page: it loads external HTML modules, injects them into the DOM, and notifies the rest of the application when everything is ready to run. Although its behavior is global—serving as a template engine

for any project—it is included within the custom structure for a fundamental reason: scalability.

As the student progresses through the exercises in the guide, each new project will incorporate its own HTML modules, routes, and visual structure. If `template-app.js` were located in a fixed, global location, it would require constant modification to import project-specific components, resulting in:

- code that is difficult to maintain,
- cross-dependencies between projects,
- a risk of breaking previous templates,
- a loss of modularity.

By keeping `template-app.js` within the project's custom folder, each project can:

- define its own HTML modules,
- control which components are loaded, adapt the template without affecting other projects,
- maintain a clean and scalable workflow.

In other words:

The file is included in the custom structure so that each project has its own independent, flexible, and extensible template engine, avoiding the need for a single global file to adapt to every future project.

This ensures that the templating system is modular, reusable, and completely autonomous within each exercise.

The flowchart of the process occurring in this file is shown in the following image:

Table 3. Flowchart for `template-app.js`

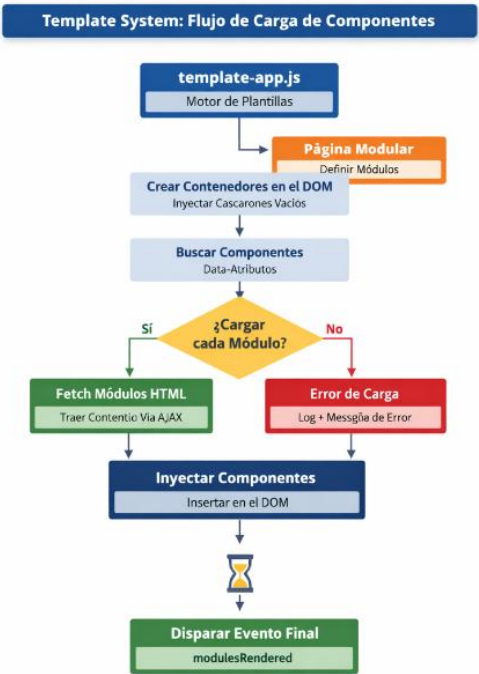


Table 3. Script for the template-app.js file

```
// File: template-app.js
function Pagina(title) {
  this.titulo = titulo || "New Web Page";
  this.head = [];
  this.body = [];
}

Pagina.prototype.agregarEstilo = function(retags) {
  this.head.push('<link rel="stylesheet" href="{rutaCss}">');
};

Pagina.prototype.agregarScript = function(rutaJs) {
  this.body.push('<script src="{rutaJs}"></script>');
};

Pagina.prototype.agregarComponente = function(rutaHtml) {
  this.body.push('<div data-
componente="{rutaHtml}"></div>');
```

```

};

Pagina.prototype.generarHtml = function() {
// We return only the internal body structure for controlled
injection
return this.body.join("\n");
};

// Derived class: PaginaModular
function PaginaModular(titulo) {
Pagina.call(this, titulo);
}

PaginaModular.prototype = Object.create(Pagina.prototype);
PaginaModular.prototype.constructor = PaginaModular;

PaginaModular.prototype.cargarModulos = function() {
this.agregarComponente("./assets/doc/index-article.html");
};
// =====
// ORCHESTRATION AND RENDERING
// =====
const pagina = new PaginaModular("My Modular Page");
pagina.cargarModulos();

// Instead of destroying the document, we inject the shells into
the existing body
const htmlEstructural = pagina.generarHtml();

// Check if the DOM is ready or still being parsed
if (document.readyState === "loading") {
document.addEventListener("DOMContentLoaded", () =>
inicializarInyeccion(htmlEstructural));
} else {
inicializarInyeccion(htmlEstructural);
}
/**
* Inserts component containers and initiates asynchronous
loading. */

function initializeInjection(htmlStructure) {

```

```

// Look for a specific container or use the body as a safe
fallback
const mainContainer = document.getElementById("dynamic-
modules-container") || document.body;

// Clean insertion that respects the browser's synchronous tree
mainContainer.insertAdjacentHTML("afterend",
htmlStructure);

// Execute the fetch process for the injected components
processDynamicComponents();
}

/** Maps network promises, resolves local files, and notifies the
system. */

function processDynamicComponents() {
const components = document.querySelectorAll("[data-
componente]");

const loadPromises = Array.from(components).map(async
(element) => {
const path = element.getAttribute("data-componente");
try {
const response = await fetch(path);
if (!response.ok) throw new Error(`HTTP ${response.status} -
${response.statusText}`);

const content = await response.text();
element.innerHTML = content;
console.log(`[Template Engine]: Component ${path}
successfully injected.`);
} catch (error) {
console.error(`[Template Engine Error]: Failed to load:
${path}`, error);
}
});

// Wait for the resolution of the array of promises
Promise.all(loadPromises).then(() => {
console.log(" Template System: All components have been

```

```
rendered. Dispatching event...");  
  
// Trigger the global custom event that app.js is waiting to hear  
const modulesReadyEvent = new  
CustomEvent("modulesRendered");  
document.dispatchEvent(modulesReadyEvent);  
});  
}
```

Main functions of this file:

- Defines a page structure using the `Pagina` and `PaginaModular` classes.
- Inserts empty containers into the DOM where components will subsequently be loaded.
- Uses `fetch()` to retrieve external HTML files and place them inside those containers.
- Waits for all modules to load and, upon completion, dispatches the `modulesRendered` event to notify `app.js` that the page is ready to initialize the application logic.

In summary:

`template-app.js` handles assembling the page in a modular and asynchronous manner, loading the necessary HTML components and notifying the rest of the system when everything is ready to run.

1.3.4. Publishing the `data-persistence` project online

To develop the data-persistence project and verify its functionality, students may choose to install the custom files independently on their own websites or integrate them into a template system. The choice between these two options is left to the individual, depending on the approach they wish to take with their work.

This flexibility allows for experimentation with both a standalone structure—ideal for understanding the complete backend and frontend workflow—and a modular, template-based architecture, which facilitates reusability and scalability in future projects.

Since the project includes both static and dynamic components, publishing it to our shared online server allows us to verify only the static portion: filling out the form, saving data to JSONLocal, and viewing the stored information in the browser. You can access this version and verify this functionality by scanning the QR code.



The dynamic component, which requires an Express server, can only be evaluated in a local environment. Later, when we explore cloud deployment, we will overcome this limitation using platforms that enable the end-to-end execution of modern applications. These cloud solutions will be examined in detail in Books 7, 8, and 9, where we will learn how to deploy dynamic projects in professional, scalable environments.

1.4. SQL

In the previous two sections, we explored two distinct approaches to data persistence, each associated with a specific type of project:

- Static site persistence using ``localStorage``

We learned to store information directly in the browser using JSON objects saved in ``localStorage``. This method is ideal for static applications, educational exercises, and prototypes that do not require a backend server.

- Dynamic site persistence using a ``users.json`` file on the server

We implemented a backend using Node.js and Express capable of reading, writing, and updating data in a local file on the server. This approach allowed us to understand how real-world persistence works in dynamic applications, where the server processes requests and manages information.

However, as projects grow and require handling large volumes of data, multiple users, relationships between entities, and complex queries, the need for a more robust and standardized system arises. This is where SQL—the universal language for managing relational databases—comes into play.

In this section, we will introduce SQL, explain how it organizes information into tables, records, and relationships, and discuss why it became the industry standard. We will also compare this model with the JSON storage systems we have already used to understand how data persistence evolves in professional applications.

SQL allows information to be organized into tables—composed of rows (records) and columns (fields)—and enables the establishment of relationships between them using primary and foreign keys. This relational model provides order, consistency, and advanced querying capabilities—features that are essential in any professional application.

1.4.1 What is SQL?

SQL (Structured Query Language) is the standard language used to manage and manipulate relational databases. Unlike the ``localStorage`` or ``users.json`` file storage methods used in the previous sections, SQL allows information to be organized in a structured, consistent manner that is optimized for complex queries and data relationships.

In a relational database, information is organized into tables, which function similarly to a spreadsheet. Each table consists of:

- Columns (fields): define the type of information stored, such as ID, name, or email.
- Rows (records): represent each individual data entry.
- Primary keys: unique identifiers that distinguish each record.
- Relationships: logical links between tables, essential for connecting information efficiently.

1.4.2. LMS menu project.

In the first module of Book 6, we worked on developing a web access control system with a standard, basic structure; we created the ``energyner`` database and the ``accesos``, ``users``, and ``access_log`` tables using SQL commands executed via the PHP connection between the frontend and backend. Consequently, many of the concepts introduced in this section will not be entirely new to students who have followed this guide step-by-step; rather, they represent a natural evolution toward a more robust and professional persistence model.

Review the **13-01-lms-menu project**, developed and published online in the First Module of this Book 6, as an applied practice of the contents studied. You can access it by scanning the QR code located on your right.



1.5. CRUD Operations in SQL

Once the fundamental structure of relational databases is understood, it is time to introduce CRUD operations—the set of essential actions for working with any data storage system. CRUD stands for:

Create → Create records

Read → Read or consult information

Update → Update existing records

Delete → Delete records

These same operations exist in SQL, expressed through specific commands that perform exactly the same function. That is why we say that CRUD and SQL are equivalent in purpose; the difference is simply that one is a general concept, while the other is its concrete implementation in relational databases.

1.5.1. INSERT — Create records

The **INSERT** statement allows new data to be added to a table. In SQL, these operations are expressed through clear, standardized statements that enable precise and secure data manipulation.

When developing the access control system in the first module of Book 6—specifically in the section "6.2. Local development of a web access control system with a basic, standard structure"—we created the ``access_insert_records.php`` file. This file was responsible for managing the registration form and executing the "Create" operation (the "C" in CRUD).

In that exercise, we implemented data insertion into the corresponding database table using PHP and SQL statements, applying a complete workflow covering validation, connection, and storage. The following code block—which we analyzed in detail in that module—

demonstrated exactly how this insertion operation was performed using prepared statements to ensure security and efficiency.

Table 1. SQL INSERT statement recording data in the "access" table of the `energyner` database.

```
if ($_SERVER["REQUEST_METHOD"] != "POST") {  
    echo "<span class='msg-info'>Record the data</span>";  
} else {  
    require __DIR__ . "../server/config/13-  
01local_db_config.php";  
    $table = "access";  
    $name = htmlspecialchars($_POST['name']);  
    $email = htmlspecialchars($_POST['email']);  
    $password = htmlspecialchars($_POST['password']);  
  
    if (!empty($name) && !empty($email) && !empty($password)) {  
  
        $sql = "INSERT INTO $table (name, email, password)  
VALUES (?, ?, ?)";  
        $stmt = $conn->prepare($sql);  
  
        if (!$stmt) {  
            die("Error preparing the query: " . $conn->error);  
        }  
  
        $stmt->bind_param("sss", $name, $email, $password);  
  
        $RESP = $stmt->execute()  
        ? "New record created successfully.."  
        : "Error executing the query: " . $stmt->error;  
  
        echo "<div class='result'>$RESP</div>";  
  
        $stmt->close();  
    } else {  
        echo "<div class='result'>Please, complete all fields</div>";  
    }  
    $conn->close();  
}
```

This block implements the complete process for receiving data from a form, validating it, and storing it in a MySQL database. First, it verifies that the request was made using the POST method; if not, it displays an informational message. When the form submits data, the script loads the connection configuration, captures the submitted values (name, email, password), and sanitizes them using `htmlspecialchars()` to prevent injections or malicious characters. It then checks that the fields are not empty before proceeding.

If the data is valid, a secure SQL statement is prepared using prepared statements, parameters are bound, and the **INSERT** command is executed to record the information in the `acceso` table. Finally, the script displays a message indicating whether the operation was successful or if an error occurred, and closes both the statement and the database connection.

1.5.2. SELECT — Query information

The SELECT statement is the foundation of all SQL queries. It allows for the retrieval of complete or filtered data, depending on system requirements.

Referring back to the exercise from Module 1 of Book 6, in the file `access_view_records.php`, we write a PHP block containing SQL statements to fetch, process, and display—in the browser—the records stored in the `access` table.

Table 1. SQL SELECT statement reading data from the "access" table in the `energyner` database.

```
<?php
require __DIR__ . "/../config/13-01local_db_config.php";

#We assign the connection to a specific database.
$OK="Conexion OK" ;

$table="access";

//echo "Tabla seleccionada=". $table;

# SQL query to select all records
$sql = "SELECT * FROM $table ";
```

```
$result = $conn->query($sql);

#Check for results and print them.
-----Rest of the code-----
```

This code snippet establishes a database connection and executes an SQL query to read all records from the `access` table. First, it imports the configuration file containing connection parameters, defines the target table, and constructs the `SELECT \* FROM access` statement. It then sends the query to the server using `\$conn->query(\$sql)` and prepares the result for processing or display.

In summary, this block accesses the database, executes a read query (a CRUD "Read" operation), and checks for the existence of records so they can subsequently be printed or manipulated according to system requirements.

1.5.3. UPDATE — Updating records

Allows you to modify existing information without having to delete the entire record.

Table 1. SQL UPDATE statement, updating data in the "access" Table of the B of D energyner

```
// Update last login
$update = $conn->prepare("UPDATEusers SET last_login =
NOW() WHERE user_id = ?");
$update->bind_param("i", $user['user_id']);
$update->execute();
```

We encountered the scope of this SQL statement while programming the `13-02login\_user.php` file during the development of the "Multi-User – Controlled Test" Access Control project—a SaaS system we built in section 6.4 of Module 1, Book 6.

This code snippet executes the CRUD "Update" operation to record the user's last login time. First, it prepares an SQL statement that modifies the `last\_login` field by assigning it to the server's current time via `NOW()`, filtering by the corresponding `user\_id`. It then binds that identifier as a secure parameter and executes the query, thereby storing the exact moment the user accessed the system in the database.

1.5.4. DELETE — Deleting records

Deletes one or more records from the table.

Table 1. Deleting a record from the table.

DELETE FROM users WHERE id = 2;

1.5.5. CRUD Y SQL in Web Applications

In a modern web application, an SQL database can synchronize with the backend via two common paths:

- PHP or
- Node.js (Express).

Although they use different technologies, both follow the same logical flow to perform CRUD operations (creating, reading, updating, and deleting records). In the case of PHP, the browser sends data from a form to the server via an HTTP request; the PHP script receives the information, validates it, constructs the corresponding SQL statement (INSERT, SELECT, UPDATE, or DELETE), and executes it against the database. It then processes the result and returns a response to the frontend for display to the user. This flow is direct, simple, and widely used in educational sites and traditional systems based on PHP and MySQL.

With Node.js and Express, the process follows the same logic but employs a more modular, API-oriented architecture. The frontend sends data to a route on the Express server; there, a controller function receives the request, validates the data, and executes the SQL statement using a connector such as `mysql2` or `pg`. The API returns a response in JSON format, which the frontend interprets to display results, update views, or generate messages. This model clearly separates the interface, server logic, and database, enabling the construction of scalable, professional-grade systems. In both scenarios—PHP or Node.js—the essential flow remains the same: Frontend → Backend → SQL → Backend → Frontend, ensuring that data can be stored, read, updated, deleted, and displayed in a secure and structured manner.

You can interact with the Access Control System—a SaaS platform featured in the first module of this book—by scanning the QR code on the right with your device's camera.



This system is more than just an academic exercise; it represents a real-world model of a professional, scalable SaaS architecture capable of managing users, validating credentials, logging activity, and synchronizing data with a remote database. By accessing the application, you can see in real time how the frontend, backend, and database integrate to provide a continuous, multi-user service accessible from any internet-connected device.

1.6. SQL and Enterprise Persistence

In the previous sections, we worked with simple persistence mechanisms based on JSON files and local storage.

These solutions are ideal for understanding the fundamentals of data persistence and for developing prototypes, educational applications, or small-scale projects. However, as an application grows and requires storing large volumes of information, managing multiple simultaneous users, and ensuring data integrity, it becomes necessary to adopt more robust technologies.

SQL-based relational databases represent the natural evolution from local storage systems to enterprise-grade platforms designed to operate securely, efficiently, and at scale. While a JSON file can store structured information, an SQL database allows for organizing data into related tables, enforcing integrity rules, controlling user access, and executing complex queries in real time.

The transition from local JSON to SQL can be illustrated as follows:

JSON Local



Basic Persistence



Data Base SQL



Business Persistence

In a simple application, information can be stored locally using JSON files or the browser's `localStorage` object. This approach is useful for testing, academic exercises, and small single-user applications. However, it presents significant limitations when it comes to sharing data among users, maintaining information consistency, or ensuring the security of stored records.

SQL databases were designed precisely to address these challenges. Thanks to their client-server architecture, they allow multiple users to access the same information simultaneously while maintaining data integrity and consistency. Furthermore, they incorporate advanced mechanisms for security, backup, disaster recovery, and performance optimization.

Prominent SQL database management systems used in professional environments include:

MySQL

MySQL is one of the most popular database engines in the world. Its ease of use, extensive documentation, and excellent integration with web technologies have made it a common choice for enterprise applications, content management systems, e-commerce platforms, and online services.

PostgreSQL

PostgreSQL is renowned for its robustness, stability, and adherence to SQL standards. It offers advanced features for handling large volumes of data, transactional processing, and demanding enterprise applications. It is currently one of the most widely used databases in cloud-native architecture and modern enterprise software solutions.

SQL Server

Developed by Microsoft, SQL Server is an enterprise platform widely used by organizations operating within the Microsoft ecosystem. It provides advanced tools for data analysis, business intelligence, security, and corporate administration.

Oracle Database

Oracle Database is one of the most powerful database solutions in the enterprise market. It is commonly used by large corporations, financial institutions, government agencies, and in environments where availability, scalability, and security are critical factors.

The choice between these engines depends on the specific requirements of each project, including factors such as data volume, available budget,

technological infrastructure, and future growth needs. However, they all share a common element: they use SQL as the standard language for defining, querying, and managing information. In the context of modern development with JavaScript, Node.js, and Express, SQL databases form the core of enterprise data persistence. Through an API, web applications can securely store, query, update, and delete information, enabling the creation of scalable systems capable of handling thousands or even millions of daily operations.

The following sections of the book will explore these technologies in greater depth, starting with MySQL as a first practical introduction to the world of relational databases, and subsequently moving on to PostgreSQL and managed cloud solutions used in professional environments.

1.7. MySQL and the phpMyAdmin panel

Before reaching this point in the book, the student has already gained practical experience with relational databases through projects completed in the first module. Those exercises involved building dynamic web applications using PHP as the server-side language and MySQL as the database management system.

During that process, students learned to:

- create databases,
- define relational tables,
- set primary keys and manage records using SQL statements.

They also developed web forms capable of inserting, querying, modifying, and deleting data stored in tables, implementing the fundamental operations known as CRUD (Create, Read, Update, and Delete).

To facilitate learning and provide a complete working environment, we used two key tools to run the projects locally:

- Visual Studio Code editor
- WampServer platform

WampServer integrates an Apache server, the PHP language, and the MySQL database engine into a single installation. This allows for the creation of a local environment that closely mirrors the setup students

will later encounter on production servers and professional web hosting services.

This local environment enabled students to deploy their applications, access them via a web browser, and see in real time how user actions resulted in database changes. Each record submitted through a web form could be immediately verified within the corresponding tables, establishing a direct link between the user interface and data persistence.

A crucial tool in this process was phpMyAdmin, the graphical administration panel included with WampServer. Through this interface, students learned how to create databases, design tables, define fields and data types, execute SQL queries, and visually observe how records are stored and organized within the system.

Using phpMyAdmin also provided a practical understanding of the inner workings of relational databases. Students have been able to examine table contents, modify record values, add new columns, delete information, and manually execute SQL statements, thereby reinforcing their understanding of the concepts covered in the theory.

This prior experience provides a solid foundation for tackling future developments that involve linking databases and tables. Moving forward, the goal will shift from simply using relational databases in local environments to understanding how to integrate these technologies into modern architecture based on Node.js, Express, REST APIs, and cloud services—while upholding the same principles of data persistence, integrity, and organization applied in their earlier projects.

1.8. NoSQL

As the Internet evolved and applications began handling millions of users, massive volumes of information, and real-time data streams, traditional relational databases started encountering limitations in certain scenarios.

To address these new challenges, a new family of technologies known as NoSQL emerged—a term that originally stood for "Not Only SQL."

The goal of NoSQL is not to completely replace relational databases, but rather to offer optimized alternatives for situations where speed, flexibility, or scalability are more important than the rigid structure of a traditional database.

NoSQL databases were designed to solve problems related to:

- Large data volumen.
- High read and write rates.
- Horizontal scalability.
- Evolving data structures.
- Distributed processing across multiple servers.

While an SQL database organizes information into related tables, a NoSQL database can store documents, key-value pairs, distributed columns, or graphs, depending on the specific problem being addressed.

Today, many of the world's leading technology platforms use NoSQL technologies to manage some of their data.

1.8.1 Differences between SQL and NoSQL

Although both models aim to store and retrieve information, there are significant differences in their architecture and design philosophy.

Table 1. SQL vs. NoSQL

SQL	NoSQL
Uses tables	Uses documents, graphs, or flexible structures
Fixed schema	Dynamic schema
Table relationships	Frequently embedded data
Vertical scaling	Horizontal scaling
High consistency	High availability
Standard SQL language	APIs and specific queries

In an SQL database, the table structure must be defined before storing information.

In a NoSQL database, each document can have a slightly different structure without requiring prior modification of the general schema. This flexibility is particularly useful when data is constantly evolving or comes from multiple sources. Let us look at SQL and NoSQL snippets that perform the same function to compare them.

Table 2. Comparison of SQL and NoSQL snippets with the same function.

SQL	NoSQL equivalent
CREATE TABLE users (id INT PRIMARY KEY, name VARCHAR(100), email VARCHAR(150));	{ "_id": 1, "name": "Juan", "email": "juan@email.com" }

In SQL we define the structure of the table before storing data. In a NoSQL system, such as MongoDB, there is no table with predefined columns. Instead, the data is stored as JSON documents, each with its own internal structure. The functional equivalent would be a document within a collection called users.

- Both represent the same record, but with different philosophies:
- SQL → tables, columns, rows, data types.
 - NoSQL → JSON documents, without the need to define columns.

Every time we register a user (or several) we need to specify which collection it is stored in.

Table 3. Ways to insert documents into B of D NoSQL

Insert documents one by one insertOne collection	Insert multiple documents in a single operation insertMany collection method
db.users.insertOne({ _id: 1, name: "John" }); db.users.insertOne({ _id: 2, name: "Ana" }); db.users.insertOne({ _id: 3, name: "Luis" });	db.usuarios.insertMany([{ _id: 1, name: "Juan" }, { _id: 2, name: "Ana" }, { _id: 3, name: "Luis" }]);

1.8.2 Document Databases

Document databases represent the most popular category within the NoSQL landscape.

Instead of storing information in tables and rows, they use documents—typically represented using JSON structures. One of the best-known examples is MongoDB, which uses the same JSON structure seen in the earlier `insertMany` collection example.

Note that this format is very similar to the JSON used previously in local persistence projects. This feature makes MongoDB a particularly attractive choice for JavaScript developers, as it allows them to work with virtually identical structures on both the client and the server.

For this reason, MongoDB is frequently used alongside:

- JavaScript
- Node.js
- Express
- React
- Angular
- Vue

1.8.3 Types of NoSQL Databases

NoSQL does not represent a single technology, but rather a complete family of specialized solutions.

Document Databases

Store JSON or BSON documents.

Examples:

- MongoDB
- CouchDB

Widely used in modern web applications and REST APIs.

Key-Value Databases

Store information using pairs:

key → value

Example:

user:1001 → Juan Pérez

The most popular solution is Redis.

Used for:

- Application caching.
- Session management.
- Temporary storage.
- Real-time processing.

Columnar Databases

Organize information by columns rather than rows.

Examples:

- Cassandra
- HBase

Ideal for distributed systems handling massive volumes of data.

Graph Databases

Represent complex relationships between entities.

Example:

User

↓ follows

User

↓ belongs to

Group

The best-known solution is Neo4j.

Widely used in:

- Social networks.
- Recommendation engines.
- Routing systems.
- Complex relationship analysis.

1.8.4 Advantages of NoSQL

NoSQL databases offer numerous advantages in specific scenarios.

Flexibility

They allow documents with variable structures to be stored without the need to redefine tables.

Scalability

They can distribute the workload across multiple servers with relative ease.

Speed

Many reading and writing operations are extremely fast due to the absence of complex relationships.

Adaptation to Big Data

They are designed to handle massive volumes of information generated by millions of users and connected devices.

1.8.5 When to Use NoSQL?

The choice of NoSQL depends on the project's characteristics.

Some scenarios where it is often an excellent choice include:

Social Networks

Platforms that manage millions of posts, comments, and user relationships.

Internet of Things (IoT)

Systems that continuously receive data from sensors and smart devices.

Messaging Systems

Applications that exchange large volumes of messages in real time.

Real-Time Applications

Platforms for monitoring, event tracking, and instant analysis.

Artificial Intelligence

Systems that process vast amounts of structured and unstructured information.

1.8.6 SQL vs. NoSQL in Modern Projects

In professional practice, the question is rarely:

"SQL or NoSQL?"

The right question is usually:

"What type of data should I store in each technology?"

For example:

Users

Billing
Payments
Contracts
↓
SQL

Financial operations require integrity, consistency, and reliable transactions.

In contrast:

Logs
Events
Telemetry
Metrics
↓
NoSQL

This data is often generated in massive quantities and requires systems optimized for mass storage and rapid queries.

Each technology addresses different problems.

For this reason, both typically coexist within the same platform.

1.9. Hybrid Architecture

Modern architecture often combine multiple persistence technologies. A typical example can be represented by the following diagram:

Figure 1. Hybrid architecture



In this architecture:

- PostgreSQL stores transactional and business data.
- MongoDB stores flexible documents.
- Redis accelerates access to frequently queried data.

This hybrid model has become a common practice in modern enterprise applications and cloud-deployed platforms.

Conclusions for Chapter 1.

In this chapter, the student has traced the natural evolution of the data persistence technologies studied throughout the module:

Local JSON

↓
Basic Persistence
↓
SQL
↓
MySQL
↓
NoSQL
↓
Enterprise Databases
↓
PostgreSQL
↓
Cloud SQL

This progression provides an understanding of how modern applications evolve from simple local storage mechanisms into enterprise-grade platforms capable of managing millions of records across distributed environments and cloud services.

The knowledge gained in this chapter will serve as a foundation for subsequently exploring PostgreSQL, Cloud SQL, cloud-native architecture, and managed persistence services—components that form the core of today's enterprise applications.

2. MODERN BACKEND DEVELOPMENT ENVIRONMENT

This chapter introduces the student to the technical ecosystem in which server-side applications are built. It explores the tools, languages, and environments that make up the modern backend, highlighting the shift from traditional models to platforms based on Node.js, Express, and cloud services. The goal is to understand how a well-configured environment enables the development of scalable, secure, and professional systems capable of integrating with SaaS architecture and efficiently meeting the demands of today's digital world.

2.1 Why Is PHP No Longer Enough?

For over two decades, PHP was one of the most widely used technologies for web development.

Well-known systems such as:

WordPress

- Joomla
- Drupal
- Magento

were built using PHP.

For many years, the classic model was:

Browser

↓

Apache Server

↓

PHP File

↓

MySQL Database

This approach enabled the development of millions of websites worldwide, and we utilized it in previous projects.

However, the evolution of the Internet created new challenges:

- Real-time applications.
- REST APIs.
- Microservices.
- Mobile applications.
- Distributed architecture.

- Cloud computing.
- Massive data processing.

The traditional model based exclusively on PHP began to show limitations.

Key Limitations

a) Blocking Processing

Each request typically requires a new process or execution thread.

When thousands of users access the system simultaneously:

- memory consumption increases
- CPU usage increases
- scalability decreases

b) Separate Frontend Ecosystem

While PHP dominated the backend, JavaScript evolved rapidly on the frontend.

Developers had to work with:

- PHP on the server
- JavaScript on the client

This created a significant technological divide.

c) Cloud Architecture

Modern platforms such as:

- Docker
- Kubernetes
- Cloud Run
- AWS Lambda

favor small, lightweight, and highly scalable services.

Node.js adapted quickly to this new landscape.

2.2 Evolution Toward Node.js

As web applications grew in complexity, a need arose:

- To use JavaScript on both the browser and the server.

This idea seemed impossible because JavaScript was originally designed to run solely within the browser. The situation changed in 2009 when Ryan Dahl introduced Node.js.

Node.js made it possible to run JavaScript outside the browser.

For the first time, it was possible to develop:

- Frontend
 - Backend
 - APIs
 - Automation tools
- using a single language.

On the frontend:

```
const name="Jhon";
```

On the backend:

```
const name="Jhon";
```

The syntax is the same.

High Performance

Node.js uses Google Chrome's V8 engine. This engine compiles JavaScript into machine code extremely quickly.

Asynchronous Architecture

It can handle thousands of simultaneous connections using events.

Ideal for:

- REST APIs
- Chat applications
- Streaming
- Cloud applications

Huge Ecosystem

There are currently millions of packages available for:

- Databases
- Security
- Authentication
- Artificial Intelligence
- Machine Learning
- External APIs

2.3 JavaScript Runtime

Before Node.js, JavaScript could only run inside a browser, which provided the execution environment. Node.js introduces a new concept: the runtime. A runtime is the environment responsible for executing a program.

Node.js includes:

- V8 engine
- Memory management
- File system
- Network communication
- Process management

Thanks to this, we can execute:

```
console.log("Hola Backend");
```

from a terminal without needing a browser.

For example, in a file named `app.js` where we write the instruction:

```
console.log("My first programNode.js");
```

we can enter the command `node app.js` in the terminal to execute that file, which then outputs "My first programNode.js".

2.4 The Terminal

The terminal is the primary tool for any backend developer. It allows for direct interaction with the operating system via commands.

Windows – Options include:

- CMD
- PowerShell
- Windows Terminal

Linux – Options include:

- Bash
- Shell
- macOS

Some of the key commands are:

Table 1. Main commands to interact with the operating system

Command	Main function
cd	Changes the directory (working path).
dir o ls	Lists files and folders in the current directory.
mkdir	Creates a new folder.
rmdir o rm -r	Deletes a folder (empty or containing files).
copy o cp	Copies files or directories.
move	Moves or renames files.
del o rm	Deletes files.
cls o clear	Clears the Terminal screen.
exit	Closes the session or the Terminal window.

- All modern infrastructure uses terminals:Git
- Docker
- Node.js
- Google Cloud
- AWS
- Azure

Mastering the terminal is an essential skill for any backend developer.

2.5 npm

npm stands for: Node Package Manager.

It is the official package manager for Node.js.

When Node.js is installed, npm is installed automatically. The functions of npm are:

- Installing libraries.
- Updating packages.
- Removing dependencies.
- Running scripts.
- Sharing projects.

Table 1. Basic npm commands

Command	Main function
npm init	Starts a project and creates the package.json file using interactive prompts.
npm init -y	Creates a package.json file automatically with default values.
npm install <package>	Installs a package and adds it to the project's dependencies.
npm install -g <package>	Installs a package globally on the system.
npm install	Installs all dependencies listed in package.json.
npm uninstall <package>	Removes an installed package.
npm update	Updates all packages to their latest allowed versions.
npm list	Displays the packages installed in the project.
npm list -g -- depth=0	Lists globally installed packages.
npm run <script>	Runs a script defined in package.json (e.g., npm run dev).
npm cache clean --force	Clears the NPM cache.
npm -v	Displays the installed version of NPM.
node -v	Displays the installed version of Node.js.

2.6 package.json

Every Node.js project has a central file called package.json. This file completely describes the application.

In the data-persistence project developed in Chapter 1 of this Book and published online, we found the need to create the package.json file and define in it the essential scripts to describe and execute the application. That was the key point where we connected the frontend with the backend, enabling the Node.js server that was listening on port 3000.

During that practice we used Visual Studio Code's integrated Terminal, learned the fundamental commands and understood how these scripts automate the developer's workflow. If you want to review the complete structure and the scripts used again, you can consult section 1.3.2. Table 1: Script and structure of the project's package.json file.

2.7 Visual Studio Code (VS Code). Installing Dependencies

Throughout the development of this guide—from the first book to this chapter—the Visual Studio Code editor has served as the central tool for writing raw code, organizing projects, and executing backend processes. Its integrated terminal enabled students to install dependencies, run local servers, and view the various generated interfaces in their browsers during practical exercises. Thanks to this continuous and progressive usage, students who have followed the entire course have gained a solid command of this modern editor, making it a natural part of their professional workflow.

A summary of the main dependencies to be installed is shown in the table below:

Table 1. Common Dependencies in Enterprise Projects

Dependency	Function
Express	Web Server
Cors	Access Control
Dotenv	Environment Variables
Pg	PostgreSQL
Nodemon	Automatic Restart
Helmet	Security
Morgan	Logs
Multer	File Upload
Jsonwebtoken	JWT Authentication
Bcrypt	Password Encryption

To conclude Chapter 2, we have set up the foundational environment that we will use throughout the rest of the book:

- Node.js as the JavaScript runtime,
 - the terminal as a professional tool,
 - npm for package management,
 - package.json for project configuration,
 - Visual Studio Code as the development environment,
 - WampServer as a local platform with an Apache server,
- and the necessary dependencies to build enterprise-grade backend applications.

The next step will be to use these tools to build our first modern API—based on Express and connected to a real database—making the leap from local exercises to services featuring professional-grade logic. From this point on, subsequent books in the Programming Step-by-Step and More series will develop increasingly creative, comprehensive, and challenging projects, solidifying a learning path focused on modern architecture and production-ready systems.

7. GENERAL CONCLUSIONS.

Developing modern web applications requires far more than just mastering a programming language. Throughout this second module, it has been demonstrated that building a professional backend with JavaScript involves understanding the interplay of various technologies, tools, and architecture working in concert to deliver scalable, maintainable, and production-ready applications.

The journey began by examining data persistence mechanisms, tracing the evolution from in-memory storage and JSON files to the use of relational and document-oriented database management systems. This progression highlighted that the choice of storage mechanism depends on the specific needs of each project and that persistence is a fundamental pillar of any enterprise-grade application.

Next, the module covered the Node.js ecosystem—a platform for executing JavaScript outside the browser—incorporating essential tools such as the terminal, npm, and Visual Studio Code, which are now staples in the daily workflows of millions of developers. Mastering these tools established a solid foundation for building modern backend applications.

Building upon this technological infrastructure, the module introduced Express.js, covering the creation of HTTP servers, the use of middleware, route handling, the implementation of HTTP methods, and REST API design—all indispensable elements for client-server communication. This process demonstrated how a well-designed API facilitates the integration of multiple clients, regardless of the frontend technology employed.

A central focus of the module was the progressive development of the `*memory-task-manager*` project. It evolved from an application that stored data solely in memory into a solution connected to a PostgreSQL database, ready for deployment in both local environments and cloud platforms. This evolution allowed the reader to experience a process closely mirroring the growth trajectory of a professional project. Additionally, the importance of separating responsibilities within an application was analyzed through modular architecture, the

application of the Model-View-Controller (MVC) pattern, the incorporation of a Service Layer, and the implementation of the Repository Pattern; this demonstrated how these techniques facilitate long-term software maintenance, code reusability, and scalability.

The module also delved into current integration and deployment practices, showcasing the use of Git for version control, GitHub as a remote repository, and Vercel as a platform for the automated publishing of web applications. Thus, the reader learned not only how to develop an application but also how to prepare a project for distribution and operation in a real-world environment.

Finally, the need to optimize and protect JavaScript code through minification and obfuscation processes was examined. It was established that while code executed in the browser can always be inspected, the use of tools like Terser and JavaScript Obfuscator significantly increases the difficulty of reverse engineering, making it a recommended practice prior to any production deployment.

Collectively, the knowledge gained in this module fosters an understanding that a modern backend application results from the integration of multiple disciplines: data modeling and persistence, API development, architectural organization, deployment automation, version control, and software protection. Rather than merely learning isolated technologies, the reader has acquired a comprehensive view of the modern web application development process, following a workflow closely aligned with those used by professional development teams.

Upon completing this second module, the reader possesses the necessary fundamentals to design, build, test, deploy, and maintain JavaScript-based backend applications using industry-standard technologies. This knowledge forms a solid foundation for tackling more advanced topics—such as distributed architecture, microservices, containers, cloud services, authentication, security, observability, and continuous integration—which represent the natural evolution of contemporary enterprise applications.

8. REFERENCES.

1. Eloquent JavaScript

Author: Marijn Haverbeke

One of the best texts for understanding:

- Modern JavaScript
- asynchrony
- Promises
- `fetch()`
- APIs
- data structures

Level: Intermediate.

2. JavaScript: The Definitive Guide

Author: David Flanagan

Reference work for:

- Advanced JavaScript
- JSON
- asynchrony
- Promises
- modern web programming

Level: Intermediate – Advanced.

3. You Don't Know JS Yet

Author: Kyle Simpson

Especially recommended for:

- Event Loop
- asynchrony
- Promises
- `async/await`
- JavaScript architecture

Level: Intermediate.

4. Learning JavaScript Design Patterns

Author: Addy Osmani

- Excellent introduction to:
- decoupled architecture
- modularity
- scalability
- patterns used in SaaS

Level: Intermediate.

5. RESTful Web APIs

Authors: Leonard Richardson, Mike Amundsen

Ideal for exploring in depth:

- REST
- HTTP
- endpoints
- resources
- API-first design

Level: Advanced.

6. Designing Web APIs

Authors: Brenda Jin, Saurabh Sahni

- Focused on:
- professional API design
- REST contracts
- interoperability

7. Building Microservices

Author: Sam Newman

Enter:

- distributed services
- decoupling
- SaaS
- scalability

8. Clean Architecture

Author: Robert C. Martin

Fundamental reference on:

- modern architecture
- separation of responsibilities
- decoupled backend

Level: Advanced.

9. Web Development with Node and Express

Author: Ethan Brown

Excellent bridge to:

- Node.js
- REST APIs
- middleware
- backend services

10. Cloud Native Patterns

Author: Cornelia Davis

Enter:

- cloud-native
- microservices
- Distributed APIs
- scalable systems

Level: Advanced.

11.MDN Web Docs JavaScript Guide

The most important documentation for:

- Modern JavaScript
- `fetch()`
- JSON
- Promises
- `async/await`

12. MDN HTTP Overview

Reference on:

- HTTP
- Request/Response
- Headers
- Status Codes

13. JSON.org

Official website for:

- JSON syntax
- structures
- data exchange

14. OpenAPI Initiative

- Preparation for:
- API documentation
- REST contracts

15. Node.js Documentation

Official base for:

- Node.js
- Modern APIs
- JavaScript backend

16. Express.js Documentation

Essential for:

- REST API development
- middleware
- routes

17. Google Cloud Architecture Center

Excellent reference for:

- cloud-native
- microservices
- distributed systems

18. Render Documentation

Preparation for:

- cloud deployment
- API hosting

19. OWASP Foundation

Essential reference on:

- web security
- authentication
- API protection
- goodpractices

20. Google Web Dev PWA Guides

Ideal for exploring in depth:

- PWAs
- Service Workers
- offline applications
- modern architecture frontend

9. MOTIVATIONAL PHRASES.

"Success is the sum of small efforts repeated day after day."

Robert Collier

"The only limit to your fulfillment tomorrow will be your doubts today."

Franklin D. Roosevelt

"It doesn't matter how slow you go as long as you don't stop."

Confucius

"The path to success and greatness is to develop what you already have."

ZigZiglar

"The secret of success lies in knowing more than others."

Aristotle Onassis

"It's not about how many times you fall, but how many times you get up."

Vince Lombardi

"Success is not the key to happiness. Happiness is the key to success. If you love what you do, you will be successful."

Albert Schweitzer

"The only way to do great work is to love what you do."

Steve Jobs

"The biggest risk is not taking any risks. In a world that's changing so fast, the only strategy that's guaranteed to fail is not taking risks."

Mark Zuckerberg

"Motivation drives us to begin, and habit keeps us going."

Jim Ryun

"It's not enough to tell the truth; you have to prove that you have it."

Baltasar Gracián

If you can't fly, run; if you can't run, walk; if you can't walk, crawl, but keep moving toward your goal."

Martin Luther King Jr

"No matter your physical age, you can always motivate your mind to face tomorrow."

From the Author

OTHER BOOKS BY THE AUTHOR

If you would like to continue exploring the *Programming Step by Step and More* series, you can access all related volumes, updates, and additional books available on Amazon through the QR code below.

All books in this series are available on Amazon.

Scan the QR code to discover more:



10. ABOUT THE AUTHOR.

After several decades dedicated to improving the energy efficiency of heating and cooling systems, I faced a major life change: retirement. Stepping away from my professional routine left me feeling disoriented and without purpose, which affected my emotional well-being. To overcome this, I sought a new activity that would keep me engaged and motivated.

I decided to share the experience and knowledge accumulated over more than forty years in the industry. This project was born with the goal of publishing articles, calculation tools, and instructional materials related to the various disciplines I worked with throughout my engineering career. My technical and computing skills were developed independently, deepening my understanding of theoretical and practical topics. This included web programming, which I used as a complementary tool to optimize project management, investments, finance, and marketing.

From firsthand experience, I can affirm that web programming is a powerful tool for solving complex problems in industrial environments, technical services, and everyday life.

My new outlook on the future—focused on the ongoing digital revolution—has helped me stay mentally active, continue learning, and explore new technologies through practice. In this fast-changing landscape, my perspective expands toward a horizon where AI not only optimizes processes but also reshapes industries and creates opportunities once unimaginable.

This instructional guide highlights the importance of programming languages as essential tools in the modern world and is designed to support people of all ages in developing their professional skills and achieving their career goals. It offers valuable resources both for those beginning their journey and for those seeking new ways to apply their experience.

The Author

PROGRAMMING STEP BY STEP AND MORE ®

An Instructive Guide

Book 6- Full-Stack Engineering with JavaScript: From Modern Frontend to Modular Architecture and Cloud Deployment.

Second Module: Modern Backend with JavaScript: Data Persistence, REST APIs, Modular Architecture, and Cloud Deployment.

Published in Paperback, Hardcover, and Digital

Miami, Florida, USA • 2026

English-language edition © 2026 René F. Ruano Domínguez

Translated from Spanish by René F. Ruano Domínguez

Originally published in Spanish as:

PROGRAMANDO PASO A PASO Y MÁS

INSTRUCTIVO – LIBRO 6.

Ingeniería Full-Stack con JavaScript: Del Frontend Moderno a la Arquitectura Modular y el Despliegue en la Nube

Segunda Parte: Backend Moderno con JavaScript: Persistencia de Datos, APIs REST, Arquitectura Modular y Despliegue en la Nube.

Copyright © 2026 René F. Ruano Domínguez

All rights reserved.

Publisher: Corporate Luxury Group, LLC