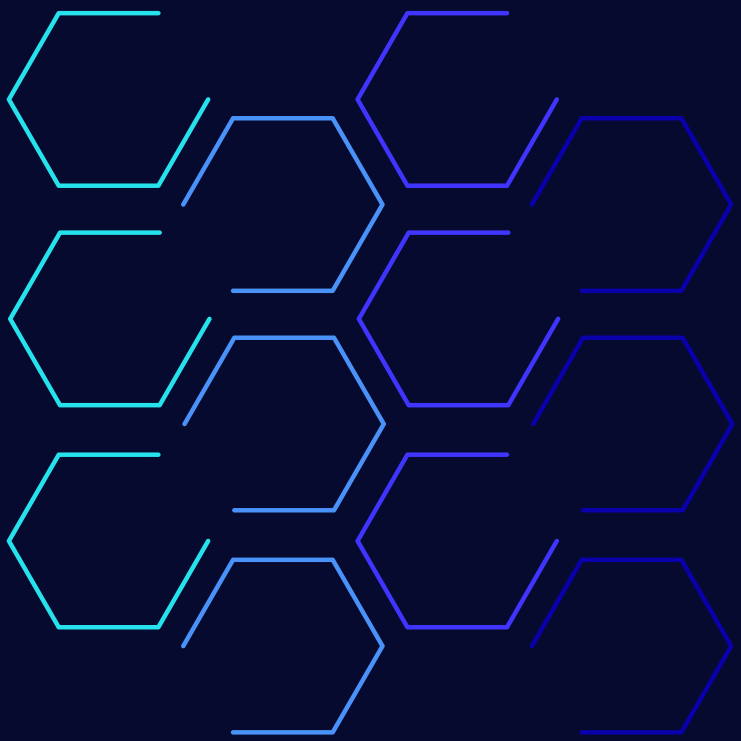
WSL FOR DELPHI AND DMVCFRAMEWORK

DEVELOP AND TEST ON LINUX FROM WINDOWS



DANIELE TETI

WSL for Delphi and DMVCFramework

Develop and Test on Linux from Windows

Daniele Teti

2026-05-29

Table of Contents

The Windows Subsystem for Linux (WSL) lets you run native Linux *distributions directly on Windows, with no heavyweight virtual machines and no dual boot. But the real superpower unlocks when you stop opening the Linux terminal by hand and start* **driving WSL from PowerShell***: launching a program with a single command, running it as a background service, monitoring it, and shutting it down cleanly when you are done.*

This guide walks you through a complete, pragmatic path: from checking the installation all the way to orchestrating Linux processes inside PowerShell scripts, covering automatic start at boot, fine-grained tuning of memory and CPU, and the techniques to **kill** *processes and distributions without corrupting state. Every concept comes with real examples and ready-to-use scripts that you will find in the* `Code/` *folder.*

Chapter 1. What's New in This Edition

This is the first edition of the guide, aligned with WSL 2 in its modern incarnation (a component distributed through the Microsoft Store, the `wsl.exe` command version 2.x with `systemd` support).

- **A "from PowerShell" approach**: every operation is expressed as a PowerShell command or script, not as an interactive session in the Linux terminal.
- **The full lifecycle**: start, management, monitoring and termination treated as one coherent flow.
- **Reusable scripts**: the `Code/` folder contains PowerShell wrappers, `wsl.conf` and `.wslconfig` examples, and scripts to start and stop services.
- **Accuracy notes**: where the online documentation is ambiguous or outdated, the guide flags the actual behavior and the most common pitfalls.

Chapter 2. Why WSL Is Essential for the Developer

Before diving into the commands, it is worth pausing on a fundamental question: why should a developer working on Windows care about Linux? Today the answer is almost always the same: **because the software you write will, sooner or later, run on Linux.**

2.1. The World Runs on Linux

Nearly all servers, containers, CI/CD pipelines and cloud services run on Linux. Even if you develop on Windows with Windows tools, the **target** environment for your code is Linux: a Docker image, a cloud machine, a Kubernetes cluster. This creates the classic "works on my machine" gap: bugs that show up only in production because Windows and Linux differ in details that matter:

- **Case-sensitive filesystem**: on Linux `Customers.json` and `customers.json` are different files; on Windows they are not. A wrong `include` or path passes the tests on Windows and then blows up in production.
- **Path separators and line endings**: `/` versus `\`, `LF` versus `CRLF`.
- **Permissions and users**: the Unix permission model does not exist on Windows.
- **System libraries and network behavior**: sockets, signals, environment variables.

WSL closes this gap by bringing a **real Linux kernel** onto your development machine. Not an emulation, not a "near-Linux": the binary you test in WSL behaves the way it will behave on the server.

2.2. What Made Everything Harder Before WSL

Historically, to keep Linux within reach a developer on Windows had three options, all with a price:

Approach	The price to pay
Dual boot	Rebooting the PC to switch operating systems: no Windows and Linux together, a broken workflow.
Full virtual machine	RAM and CPU reserved up front, slow startup, awkward file sharing, managing a second OS in every respect.
Remote / cloud server	Costs, latency, dependency on the connection, no offline work.

WSL removes these trade-offs: Linux and Windows live together on the same desktop, share the filesystem, and Linux programs can even open graphical windows (WSLg) or be invoked from a Windows script. The WSL 2 VM is lightweight and starts "lazily": it consumes resources only when you use it.

2.3. The Concrete Value for Developers

Benefit	Why it matters in daily work
Parity with production	You test your code in the same environment where it will run, immediately catching the "Linux-only" bugs.
Native toolchain	`gcc`, `make`, `bash`, `ssh`, `openssl`, `psql`, `git`, package managers: everything one command away, without rough Windows ports.
Real Docker and containers	Containers run on a true Linux kernel, just like in production.
Zero mental overhead	No reboots, no second PC: type a command and Linux is right there.
Two-way integration	From Windows you launch Linux programs; from Linux you launch Windows `.exe` files. The two worlds talk to each other.
Disposable environments	You install services (a database, a broker) in a distribution, and if it breaks you reset it with a single command.

2.4. "But I Do Desktop Development": Why It Concerns You Too

There is a very common belief among those who write desktop applications with Delphi: "I do VCL/FMX on Windows, Linux is none of my business." That was true twenty years ago. Today it is an oversimplification that, at best, makes you work with more friction and, at worst, makes you spend more.

The point is that **a modern desktop application is rarely an island**. Even the most "traditional" client almost always talks to something that, outside your machine, runs on Linux:

Component of the solution	Where it typically lives
REST API/backend the client connects to	Linux server or container (cloud, on-premise)
Database (PostgreSQL, MySQL, MariaDB)	More often on Linux than on Windows
Cache and message queues (Redis, RabbitMQ, Kafka)	Born and optimized for Linux; on Windows support is partial or absent
Auxiliary micro-services (PDF generation, image processing, scheduled jobs, integrations)	Almost always Linux containers
CI/CD, build servers, staging environments	Linux pipelines

In other words: even if **your** part is the desktop client, the **solution** around that client almost always includes Linux pieces. And this is where the problem arises for those who say "Linux doesn’t interest me."

2.4.1. What You Lose (or How Much You Pay) by Ignoring Linux

Consequence	Detail
You cannot test your dependencies locally	If your desktop client talks to an API, a Redis or a PostgreSQL, then without local Linux you have to rely on a shared server, on the cloud (which costs money) or on a colleague. Development and testing become slow and dependent on others.
You discover integration problems late	Without an environment that resembles production, "boundary" bugs (formats, time zones, encoding, network behavior) only surface at a late stage.
Architectural choices narrow	Forcing "Windows only" on the whole solution pushes you toward alternatives that are often more expensive in licenses and infrastructure. You rule out Linux containers, cheap VPSs and services born on Linux from the start.
Infrastructure costs rise	Windows hosting or equivalent server licenses generally cost more than a Linux equivalent. Not knowing how to work with Linux leads you to pick the most expensive option because it is the only one you know.
Software available only (or better) on Linux	Many tools you would run alongside your client (Redis, RabbitMQ, certain processing tools) have their reference environment on Linux. On Windows you find limited, unofficial or abandoned ports.

2.4.2. WSL Is the Answer at Almost Zero Cost

The beauty is that you do not have to give up Windows or become a Linux sysadmin. WSL lets you have **all** those Linux pieces on your development machine, right next to Delphi, without heavyweight VMs and without reboots:

- Want to verify that your FMX client correctly reads data from **PostgreSQL**? You start it in WSL with one command.
- Does your application use a **Redis** cache or a **RabbitMQ** queue? You run them locally just like in production.
- Does your solution include a small **micro-service** to generate reports or

PDFs? You develop and test it in WSL, ready to be containerized.

- Need to verify that your desktop app behaves well when the **backend** is the real one (Linux)? You run it in WSL and point the client at it.

The conclusion is not "you have to abandon the Windows desktop." It is that **"I develop on Windows, so Linux doesn't interest me" is no longer a neutral position**: it is a choice that reduces the options of your solution or increases its costs. WSL removes the alibi, because it gives you Linux **inside** Windows with no practical compromises. Knowing it widens the range of solutions you can propose and, often, lowers the final bill.

For a backend developer (and all the more so for a **Delphi developer** compiling for the Linux64 target) WSL is not a toy for the curious: it is the daily testing ground where code meets the real execution environment **before** deployment. But as we have seen, the same applies to those who do desktop development: the complete solution, today, almost always speaks Linux too.

And here comes the leap in quality this guide is about: stop using WSL only "by hand" in the terminal, and start **driving it from PowerShell** to make it repeatable and automatable.

Chapter 3. Introduction: Why Drive WSL from PowerShell?

3.1. The "I Open the Terminal and Type by Hand" Problem

Those who use WSL at first almost always follow the same ritual: they open Windows Terminal, pick the Ubuntu tab, wait for the prompt, type a command, wait, type another. It works, but it does not scale.

Imagine having to start the same development environment every morning: a PostgreSQL database, a Redis broker, and a Node build watcher. Doing it by hand means opening three terminals, remembering three command sequences, and hoping you forget nothing. And when you are done, you have to remember to shut everything down, otherwise the processes keep consuming RAM in the background.

This "manual" approach has the same problems as code that mixes logic and presentation:

- **It is not repeatable**: every start depends on your memory, not on a script.
- **It is not composable**: you cannot drop a Linux command into a PowerShell pipeline or a scheduled task.
- **It is not observable**: you do not know for sure what is running and what is not.
- **It is easy to leave "zombie" processes**: distributions and daemons stay alive consuming resources.

3.2. The Solution: WSL as a Command, Not as an Environment

The key is to stop thinking of WSL as "another computer I log into" and start thinking of it as **an executable that accepts commands**: `wsl.exe`. From PowerShell you can invoke any Linux program in a single line, without ever

seeing the bash prompt:

```
# Instead of: open terminal -> type "psql" -> Enter
# Just do:
wsl -d Ubuntu -e psql
```

And because it is a command, you can put it inside a script, a pipeline, a scheduled task or a build hook:

```
# Start the whole dev stack with a single script
wsl -d Ubuntu -u root -e bash -c "service postgresql start"
wsl -d Ubuntu -u root -e bash -c "service redis-server start"
wsl -d Ubuntu --cd ~/projects/api -e bash -c "npm run dev:watch"
```

Note the difference: no interactive session, no manual step. Everything is expressed as a command, so it is repeatable, versionable and automatable.

Driving WSL from PowerShell does not replace the interactive use of the terminal: it **complements** it. You will keep using bash for exploratory work, but you will use PowerShell for everything that has to be repeatable or automatic.

3.3. When This Approach Shines

Reproducible development environments

A single script that starts databases, queues, watchers and auxiliary services.

Build pipelines

Compiling Linux code (GCC toolchain, Node builds, containers) from a Windows script or a local CI task.

Automation at login

Starting services when you log into Windows via Task Scheduler.

Mixed Windows/Linux orchestration

PowerShell scripts that coordinate Windows and Linux tools in the same

pipeline.

Resource management

Shutting down distributions at the end of the day to free RAM, programmatically.

3.4. WSL and AI Agents: Automatic Multi-platform Orchestration

There is one more reason, increasingly relevant today, to treat WSL as a command rather than as an interactive environment: **AI development agents** (Claude Code, GitHub Copilot CLI and the like) work exactly that way. An agent does not "open a terminal and type": it invokes commands programmatically, reads their output and exit code, and decides the next step. It is the same `wsl.exe -e` model we have described.

This fits WSL perfectly. An agent running on Windows, in the **same** session and **without** human intervention, can orchestrate work on both platforms:

- compile the Delphi application on Windows with the IDE/MSBuild;
- run the **Linux64** binary and the test suite inside WSL;
- query a Linux PostgreSQL, run linters or formatters available only on Linux, build a Docker image;
- collect the results, and, if something fails, retry or do a clean **teardown**.

The reason this works is the same one that makes the approach convenient for you: because **everything is a command** (repeatable, composable, with capturable output and exit code), an AI agent can chain the steps, verify them and adjust course, exactly like the local CI script we will see later. An interactive session, by contrast, an agent would not know how to drive.

The practical benefit is considerable: the agent has a **real** Linux within reach, locally, on which to install packages, run scripts and try out builds, with no remote VM, no cloud, even offline and at zero cost. For a

Delphi/DMVCFramework project it means you can hand an agent the entire cycle **build on Windows → deploy in WSL → test on Linux → teardown**, crossing the two worlds autonomously. And because every action is a plain command, it stays readable, versionable and verifiable by a human.

The scripts in the `Code/` folder (in particular `run-linux.ps1` and the `ci-local.ps1` pattern) are designed with this in mind too: they are "agent-friendly" because they are non-interactive, parametric and have guaranteed teardown. An AI agent can call them directly.

Chapter 4. Fundamental Concepts

4.1. What WSL Is and What It Is NOT

This section is in the full version, available on Leanpub.

4.2. Distribution vs Subsystem

This section is in the full version, available on Leanpub.

4.3. The Lifecycle of a Distribution

This section is in the full version, available on Leanpub.

Chapter 5. Prerequisites and Verifying the Installation

5.1. Checking Whether WSL Is Installed

This section is in the full version, available on Leanpub.

5.2. Installing WSL (if necessary)

This section is in the full version, available on Leanpub.

5.3. Listing the Distributions

This section is in the full version, available on Leanpub.

5.4. Setting the Default Distribution

This section is in the full version, available on Leanpub.

Chapter 6. Anatomy of the `wsl.exe` Command

6.1. General Form

This section is in the full version, available on Leanpub.

6.2. The Options You Will Use Every Day

This section is in the full version, available on Leanpub.

6.3. The Crucial Difference: `-e` vs Direct Command vs `bash -c`

This section is in the full version, available on Leanpub.

Chapter 7. Launching Linux Programs from PowerShell

7.1. Essential Practical Examples

This section is in the full version, available on Leanpub.

7.2. Running a Command and Capturing Its Output in PowerShell

This section is in the full version, available on Leanpub.

7.3. Running as a Specific User

This section is in the full version, available on Leanpub.

7.4. Setting the Working Directory

This section is in the full version, available on Leanpub.

7.5. Launching Programs in the Background from PowerShell's Point of View

This section is in the full version, available on Leanpub.

7.6. Linux GUI Applications with WSLg

This section is in the full version, available on Leanpub.

Chapter 8. Managing Running Processes and Distributions

8.1. Seeing Which Distributions Are Active

This section is in the full version, available on Leanpub.

8.2. Inspecting Linux Processes from the Outside

This section is in the full version, available on Leanpub.

8.3. Monitoring Resource Usage

This section is in the full version, available on Leanpub.

Chapter 9. Terminating and Killing: Shutting Down Cleanly

9.1. The Three Levels of Termination

This section is in the full version, available on Leanpub.

9.2. Level 1: Killing a Single Linux Process

This section is in the full version, available on Leanpub.

9.3. Level 2: Terminating a Distribution

This section is in the full version, available on Leanpub.

9.4. Level 3: Shutting Down All of WSL

This section is in the full version, available on Leanpub.

9.5. The "8-Second Rule"

This section is in the full version, available on Leanpub.

Chapter 10. Automatic Program Startup

10.1. Method 1: `wsl.conf` with the `[boot]` Section

This section is in the full version, available on Leanpub.

10.2. Method 2: A Reusable PowerShell Script

This section is in the full version, available on Leanpub.

10.3. Method 3: Task Scheduler (Start at Windows Login)

This section is in the full version, available on Leanpub.

Chapter 11. Advanced Configuration

11.1. `.wslconfig`: Global Settings (Windows side)

This section is in the full version, available on Leanpub.

11.2. `wsl.conf`: Per-distribution Settings (Linux side)

This section is in the full version, available on Leanpub.

11.3. The Sections of `wsl.conf`

This section is in the full version, available on Leanpub.

Chapter 12. Real Use Cases

12.1. Development with PostgreSQL

This section is in the full version, available on Leanpub.

12.2. Development with FirebirdSQL

This section is in the full version, available on Leanpub.

12.3. Starting the Docker Daemon

This section is in the full version, available on Leanpub.

12.4. Automatic Build of a Node.js Project

This section is in the full version, available on Leanpub.

12.5. Automatic SSH Connection

This section is in the full version, available on Leanpub.

12.6. Running a Bash Script with Arguments

This section is in the full version, available on Leanpub.

Chapter 13. Applicability Scenarios for Delphi Backend Development

13.1. Why a Delphi Developer Should Use WSL

This section is in the full version, available on Leanpub.

13.2. Scenario 1: Linux Server in WSL + Windows Client

This section is in the full version, available on Leanpub.

13.3. Scenario 2: Starting/Stopping the Linux WebApp from a Windows Script

This section is in the full version, available on Leanpub.

13.4. Scenario 3: PAServer in WSL for Linux64 Deployment from RAD Studio

This section is in the full version, available on Leanpub.

13.5. Scenario 4: Reverse Proxy and HTTPS Locally

This section is in the full version, available on Leanpub.

13.6. Scenario 5: The Same Distribution as Production

This section is in the full version, available on Leanpub.

13.7. Scenario 6: Containerizing the Server (Docker in WSL)

This section is in the full version, available on Leanpub.

13.8. Map of the Scenarios

This section is in the full version, available on Leanpub.

Chapter 14. WSL in the Testing and Development Phases with DMVCFramework

14.1. Phase 1: Development Loop (Dev Inner Loop)

This section is in the full version, available on Leanpub.

14.2. Phase 2: Smoke Testing the Endpoints

This section is in the full version, available on Leanpub.

14.3. Phase 3: End-to-End Integration Tests

This section is in the full version, available on Leanpub.

14.4. Phase 3b: Client Tests on Windows with the Backend in WSL

14.4.1. When to Prefer It

This section is in the full version, available on Leanpub.

14.4.2. Examples

This section is in the full version, available on Leanpub.

14.4.3. Hybrid Orchestration

This section is in the full version, available on Leanpub.

14.5. Phase 4: Load and Performance Testing

This section is in the full version, available on Leanpub.

14.6. Phase 5: An "All-in-One" Local CI Script

This section is in the full version, available on Leanpub.

Chapter 15. Automating with AI Agents: Claude Code and Codex

15.1. The Two Execution Models

This section is in the full version, available on Leanpub.

15.2. Quick Setup

This section is in the full version, available on Leanpub.

15.3. End-to-End Example: Handing the Agent Build, Deploy and Test

This section is in the full version, available on Leanpub.

15.4. Why Agents Get Along Well with WSL

This section is in the full version, available on Leanpub.

15.5. A Note on Security

This section is in the full version, available on Leanpub.

Chapter 16. Best Practices and Troubleshooting

16.1. Best Practices

This section is in the full version, available on Leanpub.

16.2. Common Problems

This section is in the full version, available on Leanpub.

Chapter 17. WSL in Production: Recommendations

17.1. Pros and Cons

This section is in the full version, available on Leanpub.

17.2. The Recommendation

This section is in the full version, available on Leanpub.

Chapter 18. Summary of the Main Commands

18.1. Startup and Execution

This section is in the full version, available on Leanpub.

18.2. Management and Monitoring

This section is in the full version, available on Leanpub.

18.3. Termination

This section is in the full version, available on Leanpub.

18.4. Configuration

This section is in the full version, available on Leanpub.

Chapter 19. Complete Table of `wsl.exe` Options

This section is in the full version, available on Leanpub.

Chapter 20. Official Resources

This section is in the full version, available on Leanpub.

Chapter 21. Scripts Included in the `Code/` Folder

This section is in the full version, available on Leanpub.