Next steps

PHP to Rust — full chapter (all sections)

Bringing Rust to work

Rust sits beside the PHP application you already run. The PHP app keeps the pages, the session, and the database it owns.

A command line tool is the usual first step. CLI and files is that tool: arguments, files, JSON, and a test you can run in CI. A report, a file conversion, or a check that already has a PHP script is enough.

A service behind the PHP app is the next step. PHP keeps the pages. Rust answers the requests you want in a separate process. Building an API is that service.

An in-process option is lighter than a full PHP extension. PHP 7.4+ ships FFI (`ext/ffi`). You build a Rust `cdylib` that exports a `#[unsafe(no_mangle)] pub extern "C" fn` , then call it with `FFI::cdef(...)` . No PHP headers and no libclang. Many PHP teams can try that in an afternoon. Edition 2024 wraps `no_mangle` in `unsafe(...)` because the attribute is an unsafe contract with the linker.

Cargo.toml

```
 [package]
name = "shop_ffi"
version = "0.1.0"
edition = "2024"

[lib]
crate-type = ["cdylib"]
```

src/lib.rs

```
 #[unsafe(no_mangle)]
pub extern "C" fn price_cents(unit_cents: u32, quantity: u32) -> u32 {
    unit_cents.saturating_mul(quantity)
}
```

crate root

```
$ cargo build --release
```

price.php

```
<?php

$ffi = FFI::cdef(
    'uint32_t price_cents(uint32_t unit_cents, uint32_t quantity);',
    __DIR__ . '/target/release/libshop_ffi.so', // .dylib on macOS, .dll on
Windows
);

echo $ffi->price_cents(1000, 2), PHP_EOL;
```

That prints `2000`. On macOS the library is `libshop_ffi.dylib`. On Windows it is `shop_ffi.dll`. Confirm the extension is loaded with `php -m | grep FFI`.

This runs from the CLI as-is. The default is `ffi.enable=preload`, so `FFI::cdef()` only works from the CLI and from `opcache.preload` scripts. Under PHP-FPM or Apache `mod_php` it throws `FFI API is restricted by "ffi.enable" configuration directive`. In a web request, declare the bindings in an opcache preload script and use `FFI::scope()`, or set `ffi.enable=true` (not recommended in production).

A proper PHP extension is still the heavier route when you need Zend types, classes, or a PECL-shaped install. ext-php-rs binds the Zend API so a Rust function can be called as a PHP function. The guide exports `hello_world(string $name): string`, prints a PHP stub with `cargo php stubs`, and installs the shared library with `cargo php install`. The build needs PHP headers and libclang on the machine that compiles it.

Shared logic in the browser is the fourth place. WebAssembly runs one pricing function in the page and on the server.

Start with one small, low-risk tool. Measure the PHP version and the Rust version on the same input before you talk about speed. Pair on the first crate. In review, `clippy` with `-D warnings` and `rustfmt` in CI are the bar this course used. A short GitHub Actions job:

.github/workflows/ci.yml

```
 name: ci
on: [push, pull_request]
jobs:
  rust:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v7
      - uses: dtolnay/rust-toolchain@stable
        with:
          components: rustfmt, clippy
      - run: cargo fmt --check
      - run: cargo clippy --all-targets -- -D warnings
      - run: cargo test --locked
```

The costs are part of the decision. A clean build takes longer than running a PHP script, because Rust compiles. Hiring is harder while fewer people have shipped Rust than PHP. The learning curve is the one you just went through: ownership, errors, and the async runtime are the price of the first useful tool.

Teams that have written up the choice include Firecracker at AWS, Pingora at Cloudflare, and Discord's move from Go. Those posts describe their systems. A Laravel app can stay a Laravel app.

> **Watch for:** A rewrite of the whole PHP application as the first Rust project. The first win is a tool you can delete if it does not help.

> **Idiomatic Rust:** One crate, measured on the same input, reviewed with clippy and rustfmt.

Side by side

Place	PHP keeps	Rust adds	Where this course taught it
CLI tool	the web app	a binary in CI	CLI and files
Service	pages and sessions	an HTTP API	Building an API

Place	PHP keeps	Rust adds	Where this course taught it
In-process (light)	the request	a `cdylib` via FFI	PHP `ext/ffi` here — CLI/preload-only unless configured
Extension	the request	a Zend extension	ext-php-rs, described here
Browser	the page	shared logic	WebAssembly

Key takeaways

- Start with one low-risk tool beside the PHP app, and measure it on the same input.
- PHP FFI plus a Rust `cdylib` is the light in-process try. ext-php-rs is the proper extension route.
- Review with `clippy -D warnings` and `rustfmt` in CI.
- Compile time, hiring, and the learning curve you just went through are part of the decision.

What's next: The Rust ecosystem.

The Rust ecosystem

crates.io is the registry. Read the README, the license, and the recent releases before you add a crate. Download counts are a weak signal.

docs.rs builds the documentation for a published version. Use the version menu that matches `Cargo.lock`. The standard library docs are the first place to look for a type you hope already exists. `cargo doc --open` builds those pages for your crate and its dependencies, on your machine. lib.rs is another index of the same crates, grouped by category.

Clippy is the linter. This course runs it with `-D warnings`. rustfmt is the formatter. rust-analyzer is the language server in the editor. `cargo add` writes the dependency line.

cargo audit checks known vulnerabilities against the RustSec advisory database. cargo deny checks licenses, bans, and those advisories in CI.

Useful extras around the everyday loop: cargo-binstall installs prebuilt cargo tools quickly. bacon is the main save-and-rerun feedback loop (the `php artisan serve`-style habit). watchexec can do the same with `watchexec -e rs -- cargo test`. cargo-chef caches Cargo dependencies in Docker layers, which pairs with Running in Docker.

This course uses edition 2024: a new edition ships every three years and is opt-in, so old crates keep compiling. Stable Rust ships a new release every six weeks, with beta and nightly beside it.

Read a crate's guide before its trait bounds. The module page on docs.rs is the API. A trait-bound error usually means the guide's import or feature flag is missing. Axum runs on Tokio. The JSON types are serde. You swap one of them by reading the replacement, then running the tests.

Crate	What it's for	Chapter
clap	Command line arguments	CLI and files

Crate	What it's for	Chapter
serde	Derive formats from Rust types	Getting started
serde_json	JSON text	CLI and files
csv	CSV files	CLI and files
anyhow	Errors for an application	Error handling
thiserror	An error type for a library	Error handling
rayon	Data parallel iterators	Concurrency
tokio	The async runtime the API runs on	Concurrency
axum	The HTTP service	Building an API
sqlx	Postgres queries	Building an API
tower-http	CORS and request tracing	Building an API
tracing	Structured log events	Building an API
tracing-subscriber	Printing those events	Building an API
tower	One-shot requests in API tests	Building an API
http-body-util	Reading a response body in a test	Building an API
proptest	Property tests	Testing

Crate	What it's for	Chapter
criterion	Timing a hot path with `cargo bench`	Testing
assert_cmd	Running the CLI in a test	CLI and files
leptos	The browser UI	WebAssembly
wasm-bindgen	The Rust and JavaScript boundary	WebAssembly
gloo-net	HTTP from the browser	WebAssembly

> **Watch for:** Adding a crate because the download count is high. Read the README, the license, and the guide for the version you pin.

> **Idiomatic Rust:** Upgrade one crate at a time, then `cargo test` and clippy.

Side by side

	PHP	Rust
Packages	Composer and Packagist	Cargo and crates.io
Docs for the language	php.net	The standard library docs and docs.rs
Lint and format	PHP_CodeSniffer, PHP-CS-Fixer	clippy and rustfmt

Key takeaways

- crates.io, docs.rs, lib.rs, and the standard library docs are where you look a crate up.

- clippy, rustfmt, rust-analyzer, `cargo add`, cargo audit, and cargo deny are the tools around Cargo.
- The table on this page is every crate this course depended on.

What's next: Where to go from here.

Where to go from here

This course stopped on purpose before a few topics. Each one has a book.

Unsafe Rust and calling into C are the Rustonomicon. The shop API, the CLI, and the WebAssembly page are safe Rust. You can ship those without `unsafe`.

Macros start with declarative `macro_rules`. The Book's macros chapter teaches that form first. Procedural macros come after it. You have already used derive macros without writing one.

The course did not teach sockets. `std::net` is the standard library's next page, and rustls is the TLS library people reach for.

Embedded Rust assumes a microcontroller. The Embedded Book is the introduction. This course assumes a normal operating system.

The deploy this course teaches is Running in Docker: one image for the API, and Postgres beside it. Past that image, a release build uses the release profile. Registries and orchestrators are outside this course.

criterion is the benchmarking harness for a hot function. Bringing Rust to work asked you to measure a whole program on the same input. criterion is the next step when one function is the question. Testing tools already showed a `cargo bench` sketch.

Four references stay useful after the course:

- The Book
- The Async Book
- Rust by Example
- This Week in Rust

Two books that fit after this course: Zero To Production in Rust (Axum and API-focused, the natural sequel to Building an API), and Rust for Rustaceans (intermediate habits once the basics stick).

You came in writing PHP. You can read Rust, and you can put one small tool next to an application you already run. That is the course. The next tool is yours.

> **Watch for:** Opening with `unsafe` because a post did. Stay in safe Rust until a measured problem needs a boundary.

> **Idiomatic Rust:** Pick the book for the topic you actually need next. Declarative macros before procedural ones. A release profile before a new orchestrator.

Side by side

Topic this course skipped	Where to read it
Unsafe and FFI	The Rustonomicon
Macros	The Book, declarative first
Networking and TLS	`std::net` , rustls
Embedded	The Embedded Book
Benchmarks	criterion

Key takeaways

- Unsafe, macros, networking, embedded, and benchmarks each have a named next book.
- The Docker lesson is the deploy this course taught.
- The next project is a small tool on your own code.