Getting started

PHP to Rust — full chapter (all sections)

About the course

In PHP

You already ship software. PHP runs the request, Composer brings in the libraries, and you find out about a bad value when that line actually runs. None of that is wasted. This course adds Rust beside that, for the CLI, the worker, or the small service where you want the compiler to refuse a whole class of bugs before the binary exists.

Rust fits next to PHP for CLIs, queue workers, small high-throughput services, and WebAssembly. The main PHP app stays as it is.

In Rust

Rust is a compiled language. `rustc` turns your source into a native binary, and that binary is what you run. There is no garbage collector. The compiler tracks who owns each value, and who is only borrowing it, and it will not produce the binary if those rules are broken.

You do not need to throw away PHP to learn that. The useful habit is the same one you already have: name the data, pass it into a function, and handle the failure. Rust spells the types in the signature, and it checks them when you compile, not when a rare request finally hits that line.

By the end of this chapter you can:

- install Rust and confirm `rustc` and `cargo`
- tell a compile-time type error from PHP quietly coercing `"5"` into an int
- create a project with Cargo the way you create one with Composer
- pin the toolchain version for the team in `rust-toolchain.toml`
- write a tiny calculator that stops at end of input, and run `cargo test`

Libraries (`cargo new --lib` , `src/lib.rs`) wait for the modules chapter. Ownership waits for its own chapter. You will see `Option` once, in the calculator, with a note that the real lesson comes later.

How this course works

Each lesson is built the same way:

- **In PHP** — the behaviour you already know
- **In Rust** — the same idea in Rust
- **Side by side** — a short table
- **Try it** — something to run in the Playground or locally

At the end of a chapter, a quiz checks the chapter. You need **80%** to pass.

Use the **Playground** button on a lesson to open that lesson's Rust in the official sandbox. The GitHub examples repo holds example crates. Lessons that have one show a GitHub button. **Quick reference** is the PHP → Rust cheat sheet (open it from the course page or a lesson). **PDF downloads** give you a printable copy of a lesson or the whole chapter.

Prerequisites: comfort with PHP (functions, Composer, running a CLI script). No prior Rust. Roughly **1–2 hours per chapter** if you run the Try it steps.

Key takeaways

- Rust sits next to PHP. It does not replace the apps you already run.
- The compiler checks types and ownership before you have a binary.
- This chapter is install, Cargo, pinning, and one small CLI.

What's next: What Rust checks for you

What Rust checks for you

In PHP

PHP reads the file when the request arrives (or when you run `php script.php`) and interprets it. A function that asks for `int` will, by default, coerce a numeric string. That check happens as the line runs. It is not a compile step. Only `declare(strict_types=1);` at the top of that file turns the call into a `TypeError`, and even then the error is still raised at runtime, for that file only.

Memory for ordinary values is freed when the last reference goes away. `__destruct` runs then, for data that is not stuck in a cycle. It is not "the garbage collector will get to it later."

add.php

```
 <?php

function add(int $a, int $b): int
{
    return $a + $b;
}

echo add('5', 3), "\n";
```

Run on PHP 8.3, that prints `8`. The string `'5'` was coerced. Add `declare(strict_types=1);` as the first statement and the same call throws `TypeError`. The file still has to run for that to happen.

In Rust

You compile, then you run the binary. A type mismatch never becomes a coercion. This program is the whole file. It compiles, and it prints `8`.

src/main.rs

```
 fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

```
fn main() {
    let sum = add(5, 3);
    println!("{sum}");
}
```

`add("5", 3)` does not compile (`E0308` , mismatched types). There is no binary until the argument is an `i32` .

The same "fail at compile time" rule covers borrowing, with one detail that is easy to get wrong. An immutable borrow ends at its last use, not at the closing brace. This program compiles. `r1` is printed, then the mutable borrow starts, then `r2` is used. Non-lexical lifetimes are why the first borrow does not last until the end of `main` .

src/main.rs · compiles

```
fn main() {
    let mut v = vec![1, 2, 3];
    let r1 = &v;
    println!("{r1:?}");
    let r2 = &mut v;
    r2.push(4);
    println!("{r2:?}");
}
```

Use `r1` again after the mutable borrow and the compiler stops you. The error is `E0502` .

src/main.rs · must fail

```
fn main() {
    let mut v = vec![1, 2, 3];
    let r1 = &v;
    let r2 = &mut v;
    println!("{r1:?}");
    let _ = r2;
}
```

`println!` of `r1` is a later use of the immutable borrow, so `r2` is rejected. Ownership and the full set of borrow errors are a later chapter. For now, the rule is: the compiler is looking at uses, not at the line where you wrote `let` .

> **Watch for:** Reading a borrow error, look for the line where the first borrow is used again. That is the line the compiler is protecting.

Idiomatic Rust: `{sum}` inside the format string, not `"{}", sum`.

Side by side

	PHP	Rust
When types are checked	When that line runs	When you compile
`add("5", 3)` with an `int` parameter	`8`, unless that file has `strict_types`	Does not compile
When unused memory goes	When the last reference drops	When the owner ends, decided at compile time

Try it

Open the working `add` program in the Playground (the button uses that file). Change `add(5, 3)` to `add(5, "3")` and read the error (`E0308`, mismatched types). Put it back, then paste the must-fail sample and confirm the code is `E0502`.

Key takeaways

- PHP coerces `"5"` to `8` unless that file opts into `strict_types`. Rust refuses the call at compile time.
- PHP frees acyclic values when the last reference drops. It does not wait for a vague GC pause.
- A conflicting borrow is `E0502` only while the first borrow is still used.

What's next: Installing Rust

Installing Rust

In PHP

You install PHP from the operating system, Homebrew, or php.new, then you install Composer beside it. `php -v` and `composer -V` tell you what you actually have. Rust's equivalent of both tools is one installer: rustup. It installs the compiler `rustc` and the build tool `cargo`, and it can keep more than one toolchain on the machine.

In Rust

On macOS or Linux, install from the official script. Read the prompt. The default is stable, which is what you want.

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Windows uses rustup-init.exe. Install the **MSVC** toolchain and the Visual Studio Build Tools workload **Desktop development with C++** when the installer offers them — without that workload, `cargo build` fails with a linker error (`link.exe` / `lld-link` not found). After it finishes, open a **new** PowerShell or Terminal window so `PATH` includes `%USERPROFILE%\.cargo\bin`.

For the Postgres and Docker chapters later, use **WSL2** (Ubuntu) and run those lessons inside the Linux environment. rustup also works inside WSL with the same curl installer as macOS/Linux.

On Linux you need a C linker before the first build (`build-essential` on Debian/Ubuntu, or `gcc` / `clang` on other distros). On macOS run `xcode-select --install` once so the Command Line Tools provide a linker.

Codespaces: open the companion examples repo with the devcontainer so Rust, wasm targets, sqlx-cli, and Postgres are already there.

Confirm all three. `rustup` is the toolchain manager. `rustc` is the compiler. `cargo` is the build tool and package manager, in the same way Composer is yours.

```
 rustup show
rustc --version
cargo --version
```

`rustup show` should list an active toolchain of `stable-<your-target>` (for example `stable-aarch64-apple-darwin` or `stable-x86_64-unknown-linux-gnu`). If a version prints, the install worked.

Editor: any editor is fine. VS Code plus the `rust-analyzer` extension is the usual setup. Set `"rust-analyzer.check.command": "clippy"` so the editor runs the same lints as `cargo clippy`. For JetBrains, use RustRover or the JetBrains Rust plugin (they use their own engine, not rust-analyzer). Neovim works with rust-analyzer as the language server.

You do not need nightly to follow this course. The next lessons use stable only.

> **Watch for:** copying `rustup default nightly` from an old post. That changes the compiler for every project on the machine. The nightly lesson shows the per-directory command instead.

> **Idiomatic Rust:** install with rustup, then trust `rustup show`, not a distro package named `rustc` that can lag years behind.

Side by side

	PHP	Rust
Install	Homebrew, phpenv, or php.new	rustup
The package tool	Composer, installed beside PHP	`cargo`, installed with rustup
Check	`php -v` and `composer -V`	`rustup show`, then `rustc` and `cargo`

Windows (PowerShell)

Confirm the toolchain the same way:

```
 rustup show
rustc --version
cargo --version
```

Environment variables for later chapters (Postgres, sqlx) look like this in PowerShell:

```
 $env:DATABASE_URL = "postgres://shop:shop@127.0.0.1:54329/shop"
$env:SQLX_OFFLINE = "true"
```

If `cargo build` fails with a linker error after a fresh install, the MSVC Build Tools are missing — re-run the Visual Studio Installer and enable **Desktop development with C++**, then open a new shell.

Try it

Run the three commands above. `cargo --version` and `rustc --version` should report the same release.

Key takeaways

- rustup installs `rustc` and `cargo` together.
- `rustup show` is the check that you are on stable.
- rust-analyzer should check with clippy.
- On Windows, rustup-init.exe plus the MSVC C++ workload; use WSL2 for Docker/Postgres chapters.

What's next: Hello Rust

Hello Rust

In PHP

`php hello.php` reads the file and runs it. There is no separate binary. Whatever `echo` prints is the program's output. OPcache can cache the opcodes so PHP does not re-parse every request, but you still ship and run source — there is no standalone binary sitting next to the file.

hello.php

```
 <?php

echo "Hello, Rust!\n";
```

In Rust

A Rust program needs a `main` function. `println!` writes a line to stdout. The `!` means it is a macro, not a function. You can ignore that until macros have their own moment. The useful fact is that this file is not run directly.

hello.rs

```
 fn main() {
    println!("Hello, Rust!");
}
```

```
 rustc --edition 2024 hello.rs
./hello
```

On Windows the binary is `hello.exe`. `rustc` writes the executable next to the source file. That file is what you run.

The binary carries the standard library compiled in. There is no separate Rust runtime or interpreter to install on the machine that runs it. It only runs on the same OS and CPU target you compiled for (a Linux `x86_64` binary will not run on macOS ARM, and so on). Cross-compiling is a later topic; for this course, build on the machine where you will run.

`--edition 2024` matters when you invoke `rustc` yourself. Cargo puts the edition in `Cargo.toml` so you do not pass the flag each time.

> **Watch for:** treating `rustc hello.rs && ./hello` as the normal workflow. It works for one file. The moment you want dependencies, tests, or fmt, you want Cargo.

> **Idiomatic Rust:** `fn main()` is the entry point of a binary. There is no class and no framework.

Side by side

	PHP	Rust
Run the file	`php hello.php`	`rustc` writes a binary, then you run that
The process	The interpreter reads the source again	The binary is the program you run
Next step	Stay with the interpreter	`cargo` once you want dependencies or tests

Try it

Compile the file and run it. Then change the line to `println!("Hello, {name}!");` and compile. `rustc` reports `E0425` (cannot find value `name`). Then add `let name = "Rust";` above it and compile again.

Key takeaways

- `rustc` produces a binary. PHP runs the source.
- Use Cargo for anything beyond a single file.

What's next: What is Cargo?

What is Cargo?

In PHP

Composer starts a project, reads `composer.json`, writes `composer.lock`, and downloads packages from Packagist. You still run PHP yourself. Cargo is that, and it is also the thing that compiles, tests, formats, and lints. One tool, the way you wish `php` and `composer` were sometimes the same command.

In Rust

`cargo new` creates a directory, a git repo, `Cargo.toml`, and `src/main.rs`. `cargo init` does the same in the current directory. Libraries (`cargo new --lib`) are in the modules chapter.

a directory you can write to

```
 cargo new hello-cargo
cd hello-cargo
```

The generated program is small.

src/main.rs

```
 fn main() {
    println!("Hello, world!");
}
```

Cargo.toml

```
 [package]
name = "hello-cargo"
version = "0.1.0"
edition = "2024"
```

`edition = "2024"` is the language edition. It is not the compiler version. Leave it on 2024 for this course.

crate root

```
 cargo run
cargo fmt
cargo clippy
cargo build
cargo build --release
```

- `cargo run` compiles and runs `src/main.rs`.
- `cargo fmt` formats the crate. Run it before you commit.
- `cargo clippy` is the linter. Default lints are enough for this course. In a crate where you want stricter advice, you can add `#![warn(clippy::pedantic)]` at the top of `src/main.rs`; this course does not require a clean pedantic run.
- `cargo build` writes `target/debug/hello-cargo`. `cargo build --release` writes `target/release/hello-cargo`, optimised. Release is faster; debug keeps more information for backtraces. Debug builds also **panic on integer overflow**; release builds wrap. That difference is why the calculator later uses `checked_*` instead of plain `+` and `/`.

Keep warnings as warnings while you work locally. In CI, treat them as errors with `RUSTFLAGS="-D warnings"` before you build, or run `cargo clippy -- -D warnings`. An unused `Result` then fails the pipeline (`unused_must_use`) instead of scrolling past. This course points at that CI setup from here on. Do not put `warnings = "deny"` under `[lints.rust]` in `Cargo.toml` for day-to-day work: every unused variable, field, or import becomes a hard error, which fights incomplete code.

`cargo check` type-checks without writing the executable. It is the fast loop while you are fixing errors.

> **Watch for:** hand-editing a version into `Cargo.toml` for every crate. From the next lesson, `cargo add` does that.

> **Idiomatic Rust:** commit the source, not `target/`. Cargo recreates `target/` when you build.

Side by side

	PHP	Rust
New project	`composer init` (or `composer create-project` from a template)	`cargo new`
Build and run	You run `php` yourself	`cargo build` and `cargo run`
Format and lint	Extra tools	`cargo fmt` and `cargo clippy`

Try it

`cargo new` a project, change the string in `println!`, and `cargo run`. Then `cargo fmt` and `cargo clippy`. Clippy should print nothing.

Key takeaways

- `cargo new` makes a directory. `cargo init` uses the one you are in.
- `cargo run`, `cargo fmt`, and `cargo clippy` are the everyday commands.
- `cargo check` is the fast loop while fixing errors.

What's next: Managing dependencies with Cargo.toml

Managing dependencies with Cargo.toml

In PHP

`composer require monolog/monolog` writes the package into `composer.json` and the exact version into `composer.lock`. Packagist is the index. You commit `composer.lock` for an application so the next install resolves the same versions. A library often does not commit it. `require-dev` is for PHPUnit and friends, not for production.

In Rust

crates.io is Packagist. `Cargo.toml` is `composer.json`. `Cargo.lock` is `composer.lock`.

Add a crate with Cargo. Do not type the version in by hand unless you are fixing one.

crate root

```
cargo add serde --features derive
```

Cargo writes the full current version it just resolved, for example:

Cargo.toml · the dependency Cargo just added

```
[dependencies]
serde = { version = "1.0.228", features = ["derive"] }
```

A bare version is a caret requirement: `"1.0.228"` means `^1.0.228`, i.e. `>=1.0.228, <2`. The lockfile picks the exact build you compiled against.

The exact version you compiled against goes in `Cargo.lock`. Commit that file for an application, including this course's CLIs and the API you will build later. Cargo now also says libraries may commit `Cargo.lock` so contributors and CI share the same graph — that differs from the old Composer habit of leaving a library unlocked. `cargo update` moves dependencies inside the ranges in `Cargo.toml` and rewrites the lockfile. CI should build with `cargo build --locked` so a missing or edited lockfile fails the build instead of quietly resolving something new.

Dev-only tools use `[dev-dependencies]`, which is `require-dev`.

crate root

```
cargo add --dev pretty_assertions
```

Those crates are available to tests, not to the shipped binary.

> **Watch for:** committing an application without `Cargo.lock`, then wondering why two machines compiled different versions.

> **Idiomatic Rust:** `cargo add`, then commit the lockfile for anything you run. Use `--locked` in CI.

Side by side

	PHP	Rust
Manifest	`composer.json`	`Cargo.toml`
Exact versions	`composer.lock`	`Cargo.lock`
Add a package	`composer require`	`cargo add`

Try it

In the hello-cargo project, run `cargo add serde --features derive` and open `Cargo.toml` and `Cargo.lock`. You should see serde in both.

Key takeaways

- `cargo add` edits `Cargo.toml`. `Cargo.lock` records the exact build.
- Commit `Cargo.lock` for applications. CI uses `--locked`.
- `[dev-dependencies]` is Composer's `require-dev`.

What's next: Stable, nightly, and pinning a version

Stable, nightly, and pinning a version

In PHP

Teams pin PHP so CI and every laptop agree. A `.php-version` file names the version for the local switcher. `composer.json` can set `config.platform.php` so Composer resolves packages as if that PHP version is installed. The Docker image tag (`php:8.3-cli` , for example) pins the version that actually runs in CI. None of these float to whatever PHP happens to be newest.

In Rust

`rust-toolchain.toml` at the repo root pins the toolchain for everyone who builds the project. rustup reads it and installs that toolchain automatically when someone runs `cargo` in the directory.

rust-toolchain.toml

```
 [toolchain]
channel = "1.94"
components = ["rustfmt", "clippy"]
```

Use the current stable at the time you pin (check `rustc --version`); `1.94` here is only an example, not a version to copy forever. A specific stable version is the usual choice for a team. The channel `"stable"` floats: rustup moves it forward every six weeks, so two laptops can be on different compilers until both update. `1.94` stays `1.94` until you edit the file. `components` installs `rustfmt` and `clippy` with that toolchain, so `cargo fmt` and `cargo clippy` work without a separate step.

If rustup auto-install is turned off (`RUSTUP_AUTO_INSTALL=0`), the pin will not download itself. Run `rustup toolchain install 1.94` (or whatever channel you pinned) once, then `cargo` again.

Nightly is only for an unstable feature a crate's docs explicitly need. Use `rustup override set nightly` in that project's directory, and `rustup override unset` when you are done. Never run `rustup default nightly` . That changes the compiler for every project that does not pin its own toolchain. Async and procedural macros are stable

(async since Rust 1.39, procedural macros since Rust 1.30). They are not a reason to leave stable.

> **Watch for:** `channel = "stable"` in a team repo. It floats, so CI and a laptop can disagree. `rustup default nightly` copied from a gist is worse: the next `cargo build` in an unrelated project follows nightly too.

> **Idiomatic Rust:** commit `rust-toolchain.toml` with a specific stable version, including `rustfmt` and `clippy`. Nightly only when a crate's docs name an unstable feature you have decided to use.

Side by side

	PHP	Rust
Pin the version	`.php-version`, or a Docker tag	`rust-toolchain.toml`
Float to newest	Whatever PHP is installed	the channel `"stable"`
A different build	another image	`rustup override set nightly` in one directory

Try it

In the hello-cargo project, add the `rust-toolchain.toml` above. Run `rustup show` inside that directory, then again from a directory outside it. Inside, rustup reports the pinned toolchain and installs `1.94` if it is missing. Outside, you still see your default toolchain.

Key takeaways

- Pin a specific stable version in `rust-toolchain.toml`. The channel `"stable"` floats.
- rustup installs that toolchain automatically for everyone who builds the project.
- Nightly is for one directory, and only when a crate explicitly needs an unstable feature. Never `rustup default nightly`.

What's next: A CLI calculator

A CLI calculator

In PHP

`fgets(STDIN)` returns a string, or `false` at end of input. A loop that ignores `false` spins forever when the pipe closes. Integer-looking division is not integer division: `7 / 2` is `3.5`. `intdiv(7, 2)` is `3`.

calc.php

```
 <?php

$line = fgets(STDIN);
if ($line === false) {
    fwrite(STDERR, "end of input\n");
    exit(0);
}

echo '7 / 2 = ', (7 / 2), "\n";
echo 'intdiv = ', intdiv(7, 2), "\n";
```

A prompt only appears before the read if you flush stdout. `echo $prompt; fflush(STDOUT);` is the PHP version of what the Rust program does with `print!` and `flush`.

In Rust

The calculator is one binary. `prompt` prints without a newline, flushes, then reads a line. `read_line` returns `Ok(0)` at end of input. That is EOF. Returning `None` from `prompt` makes `main` stop, instead of asking again forever.

`apply` is a normal function so `cargo test` can call it without stdin. It matches on the operator and uses checked arithmetic, so overflow and divide-by-zero both return `None`. PHP 8's `match` expression is the closest equivalent.

Preview, because the Option chapter has not happened yet: `Option<i32>` is either `Some(number)` or `None`. `let Some(x) = ... else { return; }` unwraps the some-case and leaves the function on none. `parse` returns a `Result`. The same `let Ok(...) else` shape handles a bad number. Both get a full lesson later. `expect` here is only for

"stdout and stdin should work." In a library you would return the error instead.

Integer `/` truncates. `7 / 2` is `3`, not `3.5`.

src/main.rs

```
 use std::io::{self, Write};

/// Preview: `Option` is a value that is either `Some(number)` or `None`.
/// The Option chapter covers it properly. Integers divide by truncation: `7 /
2` is `3`.
fn apply(operation: &str, left: i32, right: i32) -> Option<i32> {
    match operation {
        "+" => left.checked_add(right),
        "-" => left.checked_sub(right),
        "*" => left.checked_mul(right),
        "/" => left.checked_div(right),
        _ => None,
    }
}

fn prompt(label: &str) -> Option<String> {
    print!("{label}");
    io::stdout().flush().expect("stdout should flush");
    let mut line = String::new();
    let bytes = io::stdin()
        .read_line(&mut line)
        .expect("stdin should be readable");
    if bytes == 0 {
        return None;
    }
    Some(line.trim().to_string())
}

fn main() {
    println!("Welcome to the Rust CLI calculator.");
    let Some(raw_left) = prompt("First number: ") else {
        return;
    };
    let Some(raw_right) = prompt("Second number: ") else {
        return;
    };
    let Some(operation) = prompt("Operation (+, -, *, /): ") else {
        return;
    };
    let Ok(left) = raw_left.parse::<i32>() else {
        println!("The first value should be a whole number.");
        return;
    };
    let Ok(right) = raw_right.parse::<i32>() else {
        println!("The second value should be a whole number.");
        return;
```

```
    };
    if let Some(result) = apply(&operation, left, right) {
        println!("Result: {result}");
    } else {
        println!("Use + - * or /, and do not overflow or divide by zero.");
    }
}

#[cfg(test)]
mod tests {
    use super::apply;

    #[test]
    fn adds() {
        assert_eq!(apply("+", 10, 4), Some(14));
    }

    #[test]
    fn integer_division_truncates() {
        assert_eq!(apply("/", 7, 2), Some(3));
    }

    #[test]
    fn rejects_divide_by_zero() {
        assert_eq!(apply("/", 7, 0), None);
    }

    #[test]
    fn overflow_is_none() {
        assert_eq!(apply("+", i32::MAX, 1), None);
    }
}
```

crate root

```
 cargo test
cargo run
```

`cargo test` does not need you to type numbers. `cargo run` does. Press Ctrl-D (Ctrl-Z then Enter on Windows) at a prompt and the program should exit, not spin.

> **Watch for:** `println!` for a prompt. It does flush, because stdout is line-buffered, and it puts the input on the next line. `print!` without a newline is what needs the explicit flush.

> **Idiomatic Rust:** put the arithmetic in a function that does not touch stdin, and test that function. `expect` is fine for "this stream should be open" in

`main`. It is not how a library reports a bad number.

Side by side

	PHP	Rust
Read a line	`fgets(STDIN)`	`read_line`
End of input	`false`	`Ok(0)`
Integer division	`intdiv(7, 2)`	`7 / 2` is `3`

Try it

The Playground has no stdin, so **Run** only prints the welcome line and exits. Use the Playground's **Test** action (or `cargo test` locally). You should see four passed tests. Locally you can also `cargo run`, enter `7`, `2`, and `/`. The result is `3`.

Key takeaways

- `read_line` returning 0 bytes is end of input. Stop.
- `print!` plus `flush` shows the prompt before the program waits.
- `7 / 2` is `3` for Rust integers, and `3.5` in PHP. `cargo test` checks `apply`, not `main`.

Your turn

Add a `%` operator to `apply` using `checked_rem`. Write the test first, watch it fail, then make it pass. Name the test `remainder`. The assertion is `apply("%", 7, 3) == Some(1)`.

Solution

src/main.rs · the % arm inside apply

```
"%" => left.checked_rem(right),
```

Open in Rust Playground

Also extend the operation prompt so it shows `%`: `prompt("Operation (+, -, *, /, %): ")`.

src/main.rs · new test

```
 #[test]
fn remainder() {
    assert_eq!(apply("%", 7, 3), Some(1));
}
```

Open in Rust Playground

What's next: What you learned

What you learned

In PHP

You install PHP and Composer, you run source files directly, and a typed function will coerce `"5"` unless that file says `strict_types`. Composer's lockfile is how an app builds the same way twice. Nothing in this chapter asked you to stop doing that.

In Rust

This chapter taught five things:

1. **What Rust checks** — a type mismatch (`E0308`) or a live conflicting borrow (`E0502`) stops the compile; PHP often coerces until that line runs.
2. **Install with rustup** — one installer for `rustc` and `cargo`, confirm with `rustup show`.
3. **Cargo** — `cargo new`, `cargo run`, `cargo fmt`, `cargo clippy`, debug vs release (including overflow behaviour).
4. **Dependencies and pinning** — `cargo add`, commit `Cargo.lock` for apps, pin the team toolchain in `rust-toolchain.toml` (never `rustup default nightly`).
5. **A small CLI** — read lines, stop at EOF, test pure arithmetic with `cargo test`.

`[dev-dependencies]` is Composer's `require-dev`. CI should pass `--locked`. The site does not run those tests for you. Run them locally, or use the Playground **Test** action on the calculator lesson.

Key takeaways

1. rustup installs the toolchain; pin a specific stable version for the team.
2. Cargo is the everyday tool: new, run, fmt, clippy, add, test.
3. PHP coerces by default. Rust does not compile the mismatch.
4. `Cargo.lock` and `--locked` keep builds repeatable.
5. The calculator stops at EOF, and `cargo test` covers the arithmetic.

Take the chapter quiz from the button on this page. It opens a separate page. Thirteen questions, and you need 80% to pass. After you submit, each question shows a one-line explanation.

What's next: Variables

How this course is tested

In PHP

A Laravel course that ships code samples needs more than green unit tests on the LMS. The examples themselves have to compile. This site treats the Rust fences the same way: CI runs them.

In Rust

Four jobs watch the course:

1. **snippets** — `scripts/check-snippets.py` walks every lesson markdown file. Fences marked `rust` with a `fn main` must build and run. Fences marked `compile_fail=E0xxx` must fail with that error code. Unmarked fragments are listed in a report so they can be labelled over time.
2. **crates** — the companion repo runs `cargo fmt`, `clippy -D warnings`, and `cargo test --workspace`, with Postgres 16 for the shop-api sqlx tests.
3. **wasm** — `shop-pricing` and `shop-front` build for `wasm32-unknown-unknown`, plus their native unit tests.
4. **site** — the Laravel app installs, builds the frontend, and dry-runs `course:refresh` so internal course links stay valid.

You do not need to run those jobs to learn. When something in a lesson disagrees with `rustc`, open an issue: the fence should fail CI until it matches the compiler.

Key takeaways

- Runnable fences and intentional `compile_fail` fences are checked in CI.
- The companion workspace and the WASM crates are tested on every push.
- The site build plus a course dry-run catches broken internal links.