A GUIDE FOR BACKEND DEVELOPERS



Rust

for Laravel Developers

How to stop fearing the borrow checker
and start writing servers that don't crash

```
$order->pay($amount); // throws? who knows
order.pay(amount)?; // Result<Receipt, PayError> — in the signature
```

23 chapters · end-to-end project on Axum + sqlx · examples verified by the compiler

2026

AXVILAB · GUIDES FOR BACKEND DEVELOPERS

Rust for Laravel Developers

How to stop fearing the borrow checker and start writing servers that don't crash

Vitaly Ax

First edition · November 2026 · version 1.0

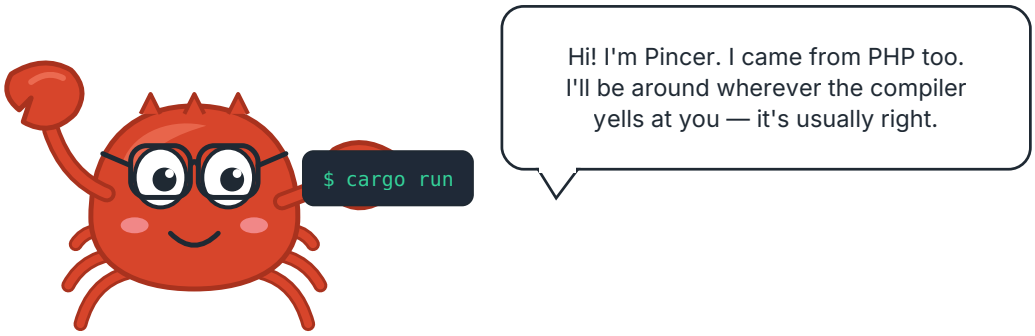
axvi.dev

© 2026 Vitaly Ax, AxviLab. All code samples are MIT-licensed. Text and illustrations — all rights reserved; redistribution without the author's permission is prohibited.

Compiler-checked: every code sample has been built and run. If something doesn't compile on your Rust version, write in — it will be fixed in the next edition.

Rust for Laravel Developers

How to stop fearing the borrow checker and start writing servers that don't crash



Who this book is for

For you, if you've been writing Laravel for a few years, know what the service container, Eloquent and queues are, and when someone says "thread" you picture Horizon, not `pthread`. You know how to program. Nobody needs to explain to you what a variable, an interface or a closure is.

You need something else: **why Rust looks the way it does, and how to map your familiar PHP reflexes onto a new language without breaking your brain.**

Most Rust textbooks are written either for people coming from C++ (who need to be told why a garbage collector exists) or for beginners (who need to be told what a loop is). You're neither. You're coming from a language where memory isn't your problem, types are optional, and a request lives for two hundred milliseconds and dies. Rust is built on the opposite assumptions, and that's exactly what makes it hard at the start. Not the syntax, but the model.

This book is about the model.

How it's structured

First, the **concept**: how Rust splits the world into data, behavior and contracts, and why it has no classes. That's the first chapter, and it's the most important one. Without it, everything else looks like a pile of random rules; with it, the rules become obvious.

Then the **language**, chapter by chapter, from variables to async. In every chapter:

- **Why you need this** — briefly, what will break or what you won't be able to write without this topic.
- **In PHP → in Rust** — how you did it before and how it's done here.
- **Why** — where the difference comes from. Every Rust "oddity" has a reason, and once you know it, it stops being annoying.
- **What the textbooks don't tell you** — the things the official docs are silent about but that bite you in the first week.
- **Try it** — an exercise. Mandatory. Ownership doesn't sink in without a keyboard.
- **Cheat sheet** — three to five lines worth remembering.

And a **project** that runs through the whole book: a small order service. It appears in the chapter on structs as a single `struct`, grows enums, traits, errors and collections, and in the penultimate chapter becomes an HTTP API on Axum with a database. Every topic lands on a familiar task right away instead of hanging in the air.

Why Rust at all, if there's PHP

Honestly: for most CRUD applications Laravel remains the better choice. Faster to write, more people, a huge ecosystem. Rust is needed where PHP hits its ceiling, and you've most likely hit it already:

- **Long-lived processes.** A WebSocket server holding thousands of connections. A worker that must not leak memory over a week. PHP was built for "request → response → die", and anything that lives longer hurts in it. Reverb on one core tops out at tens of thousands of connections; the same thing in Rust reaches hundreds of thousands on the same hardware.
- **Hosting bills.** A Rust service eats 10–30 times less memory than PHP-FPM with the same logic. One server instead of five is an invoice that arrives every month.
- **Reliability.** The compiler catches whole classes of bugs before launch: null dereferences, data races, use-after-free, a non-exhaustive `match`. What crashes in PHP at three in the morning in production simply doesn't compile here.
- **A single binary.** No interpreter, no `composer install` on the server, no extensions or PHP versions. `scp` the file and it runs. A 10 MB Docker image instead of 400.

And the main thing for you as a Laravel developer: the infrastructure around PHP is already being written in compiled languages — FrankenPHP, Caddy, RoadRunner, Octane under the hood. Knowing Rust means being able to build tools for your own ecosystem, not just use them.

What's not in this book

Embedded, Android, WebAssembly, unsafe, hand-rolled macros, Pin and Future from the inside. All of it exists and is interesting, but a web developer doesn't need it in the first year. When you do need it, you'll already know the language well enough to figure it out on your own.

Installation — through Docker, nothing on your machine

```
docker run -it --rm \
  -v rust-learn:/work \
  -v cargo-cache:/usr/local/cargo/registry \
  -w /work \
  rust:1-bookworm bash
```

Inside you get `rustc` (the compiler), `cargo` (Composer and Artisan in one) and `rustup`. The `rust-learn` volume keeps your files between runs, and `cargo-cache` keeps downloaded crates (libraries), so every new container doesn't download them again. It's the same as mounting `~/.composer/cache` into a PHP container.

If `cargo clippy` or `cargo fmt` says there's no such command, install the components once: `rustup component add clippy rustfmt`.

For quick experiments there's play.rust-lang.org: write, Run, see the result. Most exercises in this book can be done there.

Cargo is Composer + Artisan

You're used to	In Rust
`composer init`	`cargo new my-app`
`composer.json` / `.lock`	`Cargo.toml` / `Cargo.lock`
`composer require foo/bar`	`cargo add bar`
`vendor/`	`target/` (build) + `~/.cargo/registry` (cache)
packagist.org	crates.io
`php artisan serve`	`cargo run`
`php artisan test`	`cargo test`
`pint`	`cargo fmt`
`phpstan` / `larastan`	`cargo clippy`

Packages are called **crates**. After `cargo new hello`:

```
hello/
├── Cargo.toml      ← composer.json
└── src/
    └── main.rs     ← entry point
```

```
cd hello && cargo run
```

The first run compiles everything (a few seconds); subsequent runs compile only the changes.

What the textbooks don't tell you: `cargo check` instead of `cargo run`

While you're fighting the compiler, you don't need to run the program. You only need to know whether it builds. That's what `cargo check` is for: it does all the checks (types, borrowing) but doesn't generate a binary. On a big project it's many times faster than `cargo build`. The typical loop: edit → `cargo check` → edit → once it's green, `cargo run`.

The second command worth running from day one is `cargo clippy`. It's a linter like Larastan, but with advice specifically about Rust idioms: "this can be simpler", "this `clone()` is unnecessary", "write `is_empty()` instead of `len() == 0`". It has hundreds of suggestions, and almost all of them are worth following. That's how you learn to write the way everyone else writes.

One piece of advice before you start

The Rust compiler will refuse you. Often. Especially in the first week. That's not because you're a bad programmer, and not because the language is bad. It's because it checks things no language before it has checked for you.

Rust's error messages are the best in the industry: with a line number, an explanation and, most of the time, a `help: try this` hint. **Read them to the end.** Nine out of ten problems at the start are solved exactly the way the `help:` says. The compiler isn't an examiner; it's a partner who reads your code more carefully than you do.

What the textbooks don't tell you: how to read a compiler error

Here's a typical first-week error. What exactly is wrong with the code doesn't matter right now (that's the ownership chapter). Look at how the message is put together:

```
error[E0382]: borrow of moved value: `a`
 --> src/main.rs:4:16
  |
2 |     let a = String::from("hello");
  |         - move occurs because `a` has type `String`, which does not implement the
`Copy` trait
3 |     let b = a;
  |             - value moved here
4 |     println!("{a} {b}");
  |                ^ value borrowed here after move
  |
help: consider cloning the value if the performance cost is acceptable
  |
3 |     let b = a.clone();
  |              ++++++++

For more information about this error, try `rustc --explain E0382`.
```

Read it like this:

- **The first line** is the gist and the error code `E0382`. Codes are stable: google "E0382" and you'll find hundreds of explanations.
- **`-->`** is the file, line and column where the error showed up. Not necessarily where it originated.
- **Lines with `-` and `^`** are the story. The compiler shows not one spot but a chain: "created here, moved here, and here you used it". Read bottom to top. That's how you untangle the cause.
- **`help:`** is a ready-made fix, with `+++` under what you need to add. Not always the best one, but always one that works.
- **`rustc --explain E0382`** is a full article about this error with examples, right in the terminal. Works offline, in the container too.

If there are many errors, fix the **first** one. The rest are often its consequences and disappear on their own.

Let's go.

Contents

	Preface	9
1	How the language is built: the mental model	10
2	Hello, world and basic syntax	16
3	Control flow	24
4	Functions and closures	31
5	Structs — classes without classes	39
6	Enum, Option and saying goodbye to null	51
7	Traits — interfaces that can do more	60
8	Ownership — one owner for every thing	67
9	Borrowing — lending without giving away	75
10	Lifetimes — briefly and without panic	83
11	Collections and iterators — instead of PHP arrays	89
12	Error handling — Result instead of exceptions	97
13	Generics and smart pointers	106
14	Modules, crates and Cargo	113
15	Threads — concurrency PHP never had	120
16	Async — what web servers run on	128
17	Project: an order service from struct to HTTP API	138
18	Middleware, authentication and application state	165
19	Database, deeper: sqlx without Eloquent	182
20	API practices: validation, pagination, versions, documentation	195
21	Deploy: a single binary instead of Forge	214
22	Axum, deeper: layers, cookies, files and real time	227
23	Tests, macros and what comes next	267
	Glossary	277

Preface

I wrote PHP for seven years. Laravel, Eloquent, queues, Octane — I know all of it from the inside and I love it. And at the same time I hit the ceiling several times: a WebSocket server in PHP that couldn't hold the load; a worker leaking memory on its third day; a night-time incident caused by a `null` that "couldn't be there".

Before that I spent twelve years in industrial automation — Siemens, PLCs, drives. That's a different culture: everything is deterministic, types are strict, a mistake is expensive, and "it'll sort itself out somehow" doesn't work. When I opened Rust, I recognized that culture. A compiler that refuses to build code with a data race isn't an obstacle. It's the same engineering approach I'd been missing.

This book is a journal of my own transition. I wrote it while learning the language, and I tried to explain things the way I wish they had been explained to me: through what I already know from Laravel, without unnecessary theory, with examples you can run. Every "strange" Rust construct here has an answer to the question "what for", because that's exactly what I lacked in the textbooks.

The book doesn't replace the official documentation. It gives you a mental model that makes the documentation easy to read.

If you find a mistake or a place where the explanation didn't land, write to me. Every such message makes the next edition better.

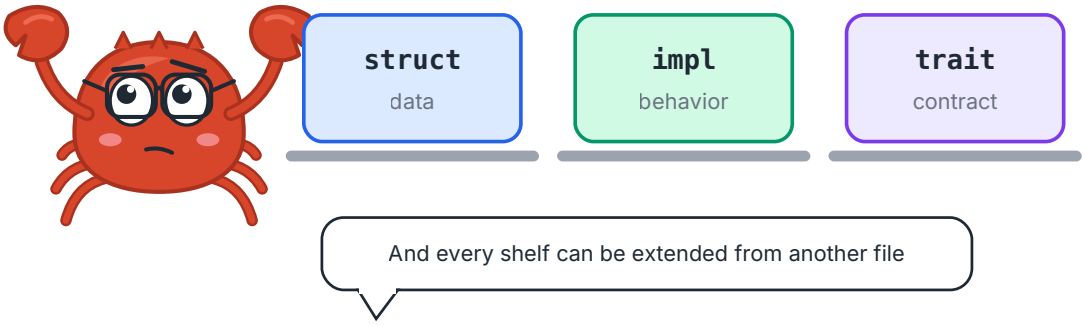
Vitaly Ax 2026

1. How the language is built: the mental model

> **Why this matters.** Without this chapter everything else is a pile of random rules. Read it first, even if you're itching to get to the code.
> **What breaks without it.** You're going to look for `class`, `extends` and `null` — and won't find them. If you don't understand what replaces them and why, every compiler error will look like arbitrary cruelty.

Code for this chapter: github.com/axvilab/rust-for-laravel-devs/tree/main/chapters/06-concept

In PHP everything is in one box. In Rust — three shelves.



Let's stop and look from above. Because if you just learn rules — "`mut` here, `&` there, `impl` over here" — they seem random, and you get lost. Once you have the model, the rules follow from it on their own.

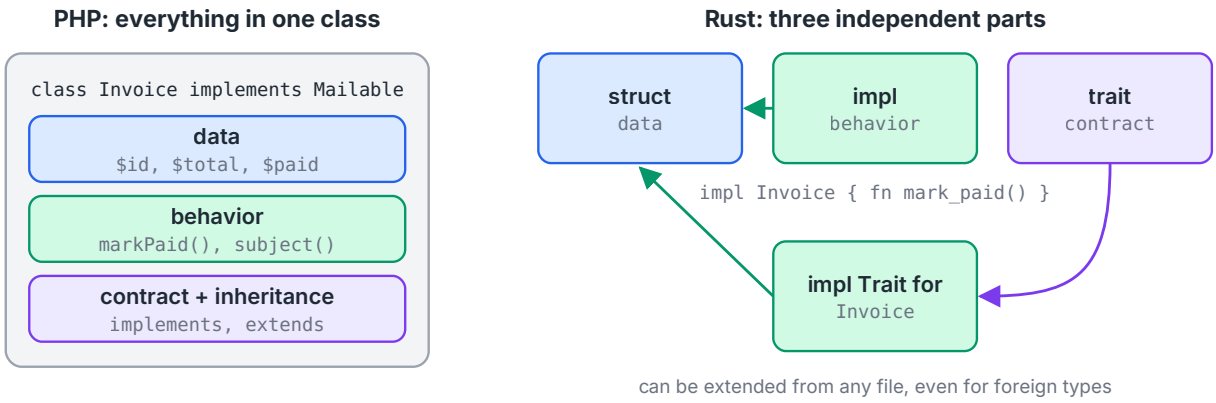
How to read the code in this chapter. We haven't covered the syntax yet; that starts in the next chapter. Here the code is only needed to show the **shape**: what lives where. Read it as pseudocode and don't try to memorize it. So that nothing looks like a cipher, here's a minimal vocabulary:

- `fn name(...)` is a function, like `function` in PHP. The return type goes after the arrow: `fn subject(&self) -> String` is `function subject(): string`.
- `id: u64` means "a field or parameter `id` of type `u64`" (an unsigned integer). The type goes **after** the name, with a colon. `f64` is a floating-point number, `bool` is the same as in PHP.
- `self` is `$this`. `&self` means the method only reads the object, `&mut self` means the method may change it. The `&` symbol is a big topic of its own; we'll get to it in the ownership and borrowing chapters.
- `format!("Invoice #{}", self.id)` is like `sprintf("Invoice #%s", $this->id)`. The exclamation mark after the name means a macro; the syntax chapter explains what that is.
- `// ...` is a comment, like in PHP.

PHP thinks in objects. Rust thinks in data and behavior separately.

In PHP (and Java, and C#) the world is built out of **objects**: a class is at the same time data (`$this->name`), behavior (`->save()`), a type (`instanceof User`) and a reuse mechanism (`extends Model`). All in one. You're used to thinking "what kind of object is this and what can it do".

In Rust and Go these things are **split into three independent parts:**



1. **Data**: `struct` or `enum`. A dumb box with fields. Can't do anything.
2. **Behavior**: functions in an `impl` block. Attached to the type, but living separately from it.
3. **Contract**: `trait`. A description of "what the type can do", with no data.

```rust
// 1. Data
struct Invoice { id: u64, total: f64, paid: bool }

// 2. Behavior specifically for Invoice
impl Invoice {
    fn mark_paid(&mut self) { self.paid = true; }
}

// 3. Contract: "anything that can be sent by email"
trait Mailable {
    fn subject(&self) -> String;
    fn body(&self) -> String;
}

// 2 again: Invoice fulfills the contract
impl Mailable for Invoice {
    fn subject(&self) -> String { format!("Invoice #{}", self.id) }
    fn body(&self) -> String { format!("Amount due: {} UAH", self.total) }
}
```

In PHP this would be a single `class Invoice implements Mailable { ... }`. In Rust it's three separate blocks that can live in different files. Why is it done this way?

Because this way you can add behavior to other people's types. Laravel has macros (`Collection::macro(...)`) because sometimes you need to add a method to a class you don't control. In Rust this is built in: write `impl MyTrait for String`, and the standard `String` gets your method. You don't inherit anything, you don't patch anything.

Think of it like this: **a struct is a row in a table. `impl` is a service that works with it. A trait is an interface.** Except there can be many services and interfaces, and you can add them later, from anywhere.

No inheritance. At all.

This is the first shock. You can't write `class Admin extends User`. There's no `parent::`, no `protected`, no abstract classes.

Why it was removed: inheritance ties data and behavior into a rigid hierarchy that later breaks. You've surely seen a `BaseController extends Controller` that forty controllers inherit from, and nobody remembers what's in it. Or a model that, through five levels of `extends`, gets a method it doesn't need. Rust (and Go) decided: better not to provide this tool at all.

What instead:

Composition: put one struct inside another.

```
struct Timestamps { created_at: u64, updated_at: u64 }

struct User { id: u64, name: String, timestamps: Timestamps }
struct Order { id: u64, total: f64, timestamps: Timestamps }
```

This is exactly what Laravel does with the `HasTimestamps` trait, only more explicit: you can see the field.

Traits with default implementations: behavior you can "plug in".

```
trait HasTimestamps {
    fn created_at(&self) -> u64;
    fn age_seconds(&self, now: u64) -> u64 {   // has a body — a default implementation
        now - self.created_at()
    }
}
```

The type implements only `created_at` and gets `age_seconds` for free. That's a PHP trait, only with type checking.

Generics: a function that works with anything that fulfills the contract. `<T: Mailable>` reads as "for any type `T` that implements `Mailable`"; details in the traits and generics chapters.

```
fn send<T: Mailable>(item: &T) {
    println!("{}: {}", item.subject(), item.body());
}
send(&invoice);
send(&newsletter);
```

In PHP you'd write `function send(Mailable $item)`. Same thing. Only in Rust it gets expanded at compile time into a separate function for each type, with no overhead for calling through an interface.

Bottom line: **everything you did with inheritance, Rust does with traits plus composition.** If you catch yourself thinking "I need a base class", what you need is a trait with default methods.

What the textbooks don't tell you: there are no constructors, just ordinary functions

In PHP an object is born through `new User(...)`, and `__construct` is a special method. Rust has no special constructor. A value is created with a literal, listing all the fields:

```
struct Invoice { id: u64, total: f64, paid: bool }

fn main() {
    let inv = Invoice { id: 1, total: 1500.0, paid: false };
    println!("{} {} {}", inv.id, inv.total, inv.paid);
}
```

And the `Invoice::new(...)` you'll see everywhere is an ordinary function that someone named `new` by convention. There can be several: `new`, `from_request`, `draft`. The consequence: an "uninitialized field" is impossible in Rust in principle. Either you've listed all the fields, or the code doesn't build. That state where half of a PHP object's properties are still `null` because someone didn't finish the constructor doesn't happen here.

Enum isn't a list of constants, it's "one of"

In PHP an enum is a list of variants. In Rust an enum is a **sum type**: a value is exactly one of the variants, and each variant can carry its own data.

Remember how many times you've written a class with a pile of nullable fields, where "if `$type === 'card'`, then `$last4` is filled, and if `'crypto'`, then `$wallet`". That's an enum, just cobbled together with no checks. Rust says: describe the variants honestly, and the compiler will make sure you've handled each one.

This is how `Option` ("there or not there") and `Result` ("worked or error") are built. They're not special language magic, just ordinary enums from the standard library. So once you understand enum + match, you'll understand half of idiomatic Rust.

What the textbooks don't tell you: a panic isn't an exception

Rust has `panic!`, an emergency stop with a message. It looks like `throw`, but it's used completely differently. Panics aren't caught in `try-catch` (formally you can, but nobody writes code like that). A panic means **a bug in the program**: an index out of bounds, "this can't happen, but it did".

Expected errors — user not found, file didn't open, malformed JSON — aren't panics. They're a `Result` value that the function returns and that you have to handle. So where in Laravel you throw a `ModelNotFoundException`, in Rust you return the error as an ordinary result. The difference seems formal, but it changes everything: the function signature shows whether it can fail, and forgetting to handle that is impossible. What this looks like in practice is in the errors chapter.

Ownership: every thing has one owner

Here's the central idea, the reason Rust looks different from everything you've seen. The next two chapters are entirely about it, but you need to hear the concept here.

In PHP memory is managed by the garbage collector. You create objects, pass them anywhere, store them in ten places, and eventually someone cleans up. You don't think about it.

Rust has no garbage collector. Instead there's a **rule: every value has exactly one owner, and when the owner goes away, the value is freed.** That's it. From this rule it follows:

- You passed a value to a function: the function became the owner, and you don't have it anymore (`move`).
- You want to keep it: let the function **look** without handing it over (`&`, borrowing).
- You want the function to change your value without taking it: lend it **with editing rights** (`&mut`), but then nobody else reads it at that moment.

It's like a library: a book belongs to the library (the owner). You can take it to read (`&`), and ten other people can read copies of the catalog along with you. But if someone took the book "for editing" (`&mut`), until they return it nobody else touches it. And when the library closes, the books are destroyed, because nobody else has them anymore.

The compiler just checks that nobody is reading a book that's already been burned, and that two people aren't editing the same page at the same time. Every "weird" borrow checker error is one of these two situations.

Types are documentation that gets checked

One last thing. In PHP you write `@param User|null $user` in a docblock and hope. In Rust a type is a claim that the compiler **proves**:

- `User`: the user definitely exists.
- `Option<User>`: it might not, and you're obliged to check.
- `Result<User, DbError>`: either it exists, or there's a specific error, and you're obliged to handle it.
- `&User`: I'm only looking, I won't change it or take it.

- `&mut User`: I will change it.
- `Vec<User>`: a list that belongs to me. `&[User]`: a list I'm looking at.

The angle brackets `<...>` here mean "what's inside": `Option<User>` is "maybe a user", `Vec<User>` is "a list of users". Like `Collection<int, User>` in PHPDoc, except the compiler checks it.

When you see a function signature in Rust, you know almost everything about it without reading the body. That's what all this "strictness" is for: not to make life harder, but so the code tells you itself what it does, and so that can't be broken unnoticed.

A quick "PHP to Rust" cheat sheet

You think in PHP	In Rust it's
a class with fields	`struct`
class methods	`impl Type { }`
an interface	`trait`
`implements`	`impl Trait for Type { }`
a PHP trait with methods	a `trait` with default implementations
`extends`	composition + traits
`abstract class`	a trait with required and default methods
`?Type` / `null`	`Option<Type>`
`throw` / `try-catch`	`Result<T, E>` + the `?` operator (the errors chapter)
`instanceof` / `switch` on type	`enum` + `match`
pass an object to a function	`&obj` (look) or `obj` (hand over)
`clone`	`.clone()` — explicit, because copying costs
garbage collector	ownership: one owner, freed on leaving scope

Next up: traits in detail, then ownership. Now you know what they're for.

Cheat sheet

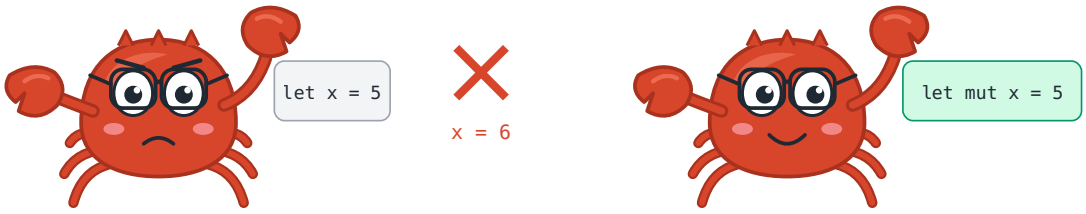
- `struct` — data. `impl` — behavior. `trait` — contract. Three separate things you can extend from any file.
- There is no inheritance. Instead — field composition and traits with default methods.
- `enum` — “exactly one of the variants, each with its own data”. `Option` and `Result` are ordinary enums.
- Every value has one owner. `&` — look, `&mut` — edit, nothing — give away.

2. Hello, world and basic syntax

> **Why this matters.** Half of the "weird" errors in the first week come from `mut`, `String` vs `&str`, and a semicolon at the end of a function.
> **What breaks without it.** You'll write `let x = 5; x = 6;` and get refused. You'll return a value from a function with a semicolon — and the function will return "nothing". You'll pass a literal where a `String` is expected — and won't understand why it doesn't fit.

Code for this chapter: github.com/axvilab/rust-for-laravel-devs/tree/main/chapters/01-basics

Immutable by default. Want to change it — say so out loud.



mut isn't a punishment, it's a hint to the reader: "this value changes here"

Your first program

```rust
fn main() {
    println!("Hello, Laravel developer!");
}
```

What matters here:

- `fn main()` is the entry point. PHP doesn't have one; a script runs top to bottom. In Rust, a program starts at `main`.
- `println!` has an exclamation mark. It's a **macro**, not a function. For now: anything with an exclamation mark is a macro, and it can do things a regular function can't (for example, take a variable number of arguments and check the format string at compile time). More on that later.
- The semicolon is mandatory, like in PHP. But there's an exception; more on it below.

Little things you'll see everywhere

Before going further, here are a dozen things that will show up in every example starting on the next page. Each one gets a detailed explanation in its own chapter; here you get exactly enough not to trip over them.

`//` **is a comment.** Same as PHP. `/* */` works too. `///` is a doc comment; more on that later.

`::` means "inside". Double colon is a path to something in a namespace, like `\` or `::` in PHP. `String::from("...")` is the function `from` that lives in the type `String`. `std::collections::HashMap` is the type `HashMap` in the `collections` module of the standard library `std`. When you see `Something::something`, read it as "something from Something".

`use` is an import. `use std::collections::HashMap;` is the same as `use App\Models\User;` in PHP: now you can write just `HashMap` instead of the full path. Everything in `std::` (the standard library) is available without installing anything, but most of it has to be imported.

An exclamation mark means a macro. `println!`, `format!`, `vec!`, `panic!` all have an exclamation mark. They're not functions but macros: pieces of code that expand at compile time. For you, right now, there's one difference: a function can't take a variable number of arguments and check a format string before the program runs, and a macro can. When you see `!`, think "built-in convenience" and move on. You won't have to write your own macros.

`vec![1, 2, 3]` is a list. A macro that creates a `Vec` (a vector, a dynamic array) from the listed elements. It's `[1, 2, 3]` in PHP. Details in the collections chapter.

Three ways to make a `String`. `String::from("text")`, `"text".to_string()`, `"text".into()` all do the same thing: turn a literal into a full-fledged `String`. (You'll also meet a fourth one, `"text".to_owned()`, with the same result.) Why a literal isn't a `String` yet, I explain a bit further down in this chapter. For now: they're all interchangeable, and the book uses all of them, because real code does too. The fine points of when each one is more convenient are in the structs chapter.

`&` means "let me look at it". An ampersand before a variable, `&name`, `&users`, means "I'm passing not the thing itself but access to it; it stays mine". This is the central idea of the language, and it gets two chapters (ownership and borrowing). Until then, just write `&` wherever it appears in the examples, and the code will work.

`Some(x)` / `None` mean "there is" / "there isn't". Rust has no `null`. Instead, a value that might be missing is wrapped: `Some(5)` means there's a five, `None` means nothing. You'll see it in a few examples before the `Option` chapter, where it's explained in full.

`.unwrap()` means "unpack it, and crash if it's empty". Many functions return not a value but "a value or an error". `.unwrap()` says: get the value, and if there's an error, stop the program with a message. That's fine in learning examples. Real code has a better way; see the errors chapter. For now: `.unwrap()` = "I'm sure everything's fine here".

`_` means "I don't care". An underscore instead of a variable name means "there's a value, but I don't need it". `let _ = f();`, `for _ in 0..10` (`0..10` is a range from 0 to 9; more on it in the next chapter), `match x { _ => ... }`. A variable that starts with an underscore (`_unused`) is the same thing, but with a name for readability.

`#[something]` is an attribute, a note for the compiler. A line in square brackets with a hash above a struct or a function isn't code that runs. It's a note attached to the next item: "compiler, pay attention to this". Same idea as PHP 8 attributes: `#[Route('/users')]` above a controller method. Most often you'll see `#[derive(Debug)]` above a struct. That's a request: "compiler, write the code for me that can print this struct with `{:?}`". Attributes and `derive` are covered in full in the structs chapter; until then, copy them along with the example.

`::<type>` is the turbofish. Sometimes the compiler can't guess a type and asks for a hint: `"42".parse::<i32>()` means "parse it as i32". The `::<>` shape looks like a little fish, hence the name. You'll see it a few times before the collections chapter, where it's explained.

That's it. Now no example contains anything you haven't seen.

Variables

In PHP:

```
$count = 5;
$count = $count + 1;   // can be overwritten
```

In Rust:

```
let count = 5;
count = count + 1;     // ERROR: cannot assign twice to immutable variable
```

By default, variables in Rust are **immutable**. If you want to change one, say so explicitly:

```
let mut count = 5;
count += 1;            // ok
```

Why. You've probably noticed this yourself in PHP: most variables are assigned once. Rust just makes that the default rule and makes mutation the exception you have to mark. When you see `mut`, you know: the value will change here, look more carefully. That's readability for free.

What the textbooks don't tell you: yellow warnings are messages too

Besides errors, the compiler emits warnings. The code builds, but rustc says "something looks suspicious here". The most common ones at the start:

```
fn main() {
    let x = 5;      // warning: unused variable: `x`
    let _y = 6;     // silence: "I know I'm not using it"
}
```

The compiler will even give you a hint: `help: if this is intentional, prefix it with an underscore: _x`. A leading underscore in a name is how you say "this is on purpose". Same with a `mut` you didn't need: `warning: variable does not need to be mutable`. Remove the `mut`.

While experimenting you can get dozens of warnings, and they get in the way of seeing the real errors. In that case you can put `#![allow(unused)]` on the first line of the file. It's an attribute meaning "don't warn about unused things in this whole file" (what attributes are: see the little things above; in detail, the structs chapter). Just don't leave it in production code: a warning about an unused variable often means a real bug. You computed a value and forgot to use it.

There's also `const`, like in PHP, for real constants, and the type is mandatory:

```rust
const MAX_CONNECTIONS: u32 = 10_000;   // underscores for readability
```

Types

PHP is dynamic: `$x = 5; $x = "five";` is fine. Rust is **static**: a variable's type is known at compile time and doesn't change. But most of the time you don't need to write it; the compiler infers it:

```rust
let a = 5;          // i32 (signed integer, 32 bits)
let b = 5.0;        // f64 (floating point, 64 bits)
let c = true;       // bool
let d = 'x';        // char — a single Unicode character, in single quotes
let e = "text";     // &str — a string (strings get their own section, there's a catch)
```

If you need to, you specify it explicitly:

```rust
let port: u16 = 8080;
```

The main numeric types are the same as in PDO/MySQL, just with shorter names:

Type	What it is	Analogy
`i8` `i16` `i32` `i64` `i128`	signed integer, N bits	`TINYINT` … `BIGINT`
`u8` `u16` `u32` `u64` `u128`	unsigned integer	`UNSIGNED`
`isize` `usize`	pointer-sized (64 bits on a 64-bit OS); `usize` is for indexes and lengths	—
`f32` `f64`	floating point	`FLOAT` / `DOUBLE`

By default an integer is `i32` and a float is `f64`. Array indexes and lengths are always `usize`; the compiler will remind you.

Important: Rust doesn't do implicit conversions. In PHP `5 + 2.5` = `7.5`. In Rust:

```rust
let x: i32 = 5;
let y: f64 = 2.5;
let z = x + y;              // ERROR: cannot add f64 to i32
let z = x as f64 + y;       // ok: converted explicitly
```

`as` is the numeric conversion operator: `x as f64` means "take `x` and make an `f64` out of it". The equivalent of `(float) $x` in PHP.

Why. Half of the "weird" bugs in PHP come from implicit conversions: `"10" == "1e1"`, `0 == "a"`, `null + 1`. Rust removes this whole category of errors: if you want to mix types, say exactly how.

What the textbooks don't tell you: overflow and the silent `as`

In PHP, an integer that doesn't fit into 64 bits quietly becomes a `float`. In Rust the type is fixed, and a `u8` greater than 255 doesn't exist. What happens on overflow depends on the build mode:

- **debug** (`cargo run`): a panic, `attempt to add with overflow`. The program crashes, and you see the bug right away.
- **release** (`cargo run --release`): no check, the value "wraps around": `255 + 1` gives `0`. Fast, but silent.

If the values are already known at compile time (`let x: u8 = 255; x + 1`), rustc won't even build the code. But if the number came from outside, from a request or from the database, explicit methods help:

```
fn main() {
    let stock: u32 = "3".parse().unwrap();   // a value "from outside"
    let sold: u32 = 5;

    // let left = stock - sold;   // debug: panic, release: 4294967294

    println!("{:?}", stock.checked_sub(sold));  // None — "didn't work"
    println!("{}", stock.saturating_sub(sold)); // 0 — clamped to the limit
    println!("{}", stock.wrapping_sub(sold));   // 4294967294 — wrapped around
    println!("{:?}", stock.checked_add(1));     // Some(4)

    println!("{}", 300_i32 as u8);    // 44 — as silently truncates
    println!("{}", -1_i32 as u32);    // 4294967295
    println!("{}", 3.99_f64 as i32);  // 3 — the fractional part is dropped
}
```

For money, stock levels and counters, use `checked_*` (returns `Some` / `None`, and you decide what to do) or `saturating_*` (doesn't go below zero). And `as` between numbers **never panics and checks nothing**. It just truncates. A safe conversion with a check is `u8::try_from(x)`; more on it in the errors chapter.

Strings: `String` and `&str`

The first thing that confuses everyone coming from PHP. Rust has **two** string types:

- `&str` is a "view" of a string that already lives somewhere (in the program's code, in another variable). Immutable. Literals like `"text"` are `&str`.
- `String` is a string that you own. It lives on the heap, and you can change it and grow it.

```
let greeting = "Hello";                    // &str, baked into the binary
let mut name = String::from("Vitalii");    // String, on the heap
name.push('!');                            // can be changed (push for one char, push_str
for a string)
let full = format!("{greeting}, {name}"); // String
println!("{full}");
```

A rule that's enough for the first while: **passing to a function, use `&str`; storing in a struct, use `String`.** You'll understand why in the ownership chapter; for now, just remember it.

Strings in Rust are always valid UTF-8. `.len()` gives the length in bytes, `.chars().count()` gives it in characters:

```
let s = "Дніпро";
println!("{}", s.len());             // 12 (bytes)
println!("{}", s.chars().count());   // 6 (characters)
```

So Rust is like PHP with `mb_*` always on and no way to turn it off.

Formatting output

`println!` and `format!` use placeholders in curly braces:

```
let name = "Vitalii";
let age = 35;

println!("{} is {} years old", name, age); // positional
println!("{name} is {age} years old");     // named (since Rust 2021)
println!("{:?}", vec![1, 2, 3]);           // {:?} — "debug" output, like var_dump
println!("{:#?}", some_struct);            // {:#?} — the same, but with line breaks
println!("{:>8}|{:<8}|", "right", "left"); // alignment
println!("{:.2}", 1.23456);                // 1.23
```

Modifiers work with names too: `println!("{users:?}")`, `println!("{price:.2}")`. Variable name, colon, then the format.

`{:?}` is your `var_dump()`. It works with all built-in types; for your own structs you'll need to add the `#[derive(Debug)]` attribute above them (see the little things at the start of the chapter; in full, the structs chapter).

What the textbooks don't tell you: the real `dd()` is `dbg!`

There are three macros for debugging, and they're not interchangeable:

- `println!` writes to standard output (stdout). For what the program shows the user.
- `eprintln!` writes to the error stream (stderr). For logs and diagnostics: it doesn't mix with the useful output if that's redirected to a file or another program.

- `dbg!` is for debugging. It prints to stderr **the file name, the line, the expression itself and its value**, and passes the value on. So you can stick it in the middle of an expression without rewriting anything:

```
fn main() {
    let price = 100;
    let total = dbg!(price * 3) + 1;   // dbg! passes the value on
    eprintln!("this goes to stderr");
    println!("{total}");
}
```

It prints `[src/main.rs:3:17] price * 3 = 300`, then `301`. That's closer to `dd()` than `{:?}`: you can see where the line came from, and you don't need to invent a label. One difference: `dbg!` doesn't stop the program. And don't forget to remove it before committing: `cargo clippy -- -W clippy::dbg_macro` will find all the forgotten ones.

Expressions and the semicolon

Here's that exception. In Rust almost everything is an **expression**, meaning it returns a value. A code block `{ ... }` returns the value of its last expression if there's **no** semicolon after it:

```
let x = {
    let a = 10;
    let b = 20;
    a + b          // no ; — this is the block's value
};
println!("{x}");   // 30
```

A semicolon turns an expression into a statement and "eats" the value. So in functions:

```
fn square(n: i32) -> i32 {
    n * n          // no return needed
}

fn square_verbose(n: i32) -> i32 {
    return n * n;  // this works too, but idiomatic code has no return
}
```

If you accidentally put `;` after `n * n`, you'll get an error: "the function should return i32, but it returns ()". `()` means "nothing", the equivalent of `void`. When you see this error, look for an extra semicolon.

Try it

On play.rust-lang.org:

1. Create a `let mut` variable with a product price (`f64`) and a quantity (`u32`). Compute the total. Notice that the compiler will demand a type conversion.
2. Print the result with two decimal places using `{:.2}` .
3. Try removing `mut` and assigning a new value to the variable. Read the error message. The Rust compiler writes very detailed messages, often with a "try this" hint. Get used to reading them; they're your main learning tool.
4. Wrap the total computation in `dbg!(...)` without changing the rest of the line. See what gets printed and in what order relative to `println!` .
5. Create `let stock: u8 = 250;` and add `10` to it with `checked_add` and with `saturating_add` . Print both results with `{:?}` . Then try `stock + 10` directly and read what the compiler says.

Cheat sheet

- Variables are immutable by default: `let mut` if you intend to change it.
- No implicit type conversions: `x as f64` .
- In function parameters — `&str` , in struct fields — `String` .
- The last line without `;` is the value of the block or function. An extra `;` = return `()` .
- `{:?}` is your `dd()` ; it needs `#[derive(Debug)]` .

That was the free sample

The full book has the remaining chapters, a project from zero to deploy, exercises with solutions, and all the code in a repository.

Full book: Rust for Laravel Developers
leanpub.com/rust-for-laravel-developers