

# Writing nginx Modules in Rust

Yugo Horie

Edition of 2026-10-05

Covers ngx-rust main as of September 17, 2026 (f5aacab) and nginx 1.30.x

Copyright © 2026 Yugo Horie

Translated from the author's Japanese original with the help of an AI model

Free sample: the full book is available at [leanpub.com/nginx-rust](https://leanpub.com/nginx-rust)

# Contents

- Getting Started
  - [1. Introduction](#)
  - [2. Setting Up the Environment](#)
  - [3. Running Your First Module](#)
- Part 1 — The Rust You Need to Read ngx-rust
  - [About Part 1](#)
  - 4. Ownership and Borrowing — Who Cleans Up a Value? (full book)
  - 5. Expressing It in Types — Option, Result, Traits, Lifetimes (full book)
  - 6. unsafe — Writing What the Compiler Cannot Guarantee (full book)
- Part 2 — Writing Modules
  - About Part 2 (full book)
  - 7. Anatomy of a Module — Taking hello's 100 Lines Apart (full book)
  - 8. Phases — Where to Hook into Request Processing (full book)
  - 9. Requests and Responses (full book)
  - 10. Where Values Live — The Allocator as a Type Parameter (full book)
  - 11. Shared Memory — Keeping Data Across Workers (full book)
  - 12. Synchronizing Processes — Locks and Zone Initialization (full book)
  - 13. Filters — Transforming Other Modules' Output (full book)
  - 14. Load Balancers — Choosing a Backend (full book)
- Part 3 — Building a Real Module
  - About Part 3 (full book)
  - 15. Counting Server Zones — A Minimal Aggregation Module (full book)
  - 16. Counting Upstreams — Without Missing Retries (full book)
  - 17. Exposing Prometheus Metrics (full book)
  - 18. Writing Tests — Making Sure the Numbers Add Up (full book)
  - 19. Pitfalls (full book)
  - 20. When ngx-rust Is Missing Something (full book)
  - 21. Case Study — What ngx\_vts Got Right and Wrong (full book)
- Appendices
  - A. Where Things Live in the ngx Crate (full book)
  - B. Glossary (full book)
  - C. How Red-Black Trees Work (full book)
  - D. Trait Index (full book)

# Chapter 1: Introduction

## Note

**About this edition.** This English edition was translated from the author's Japanese original, [Rustで書くnginxモジュール](#), with the help of an AI model. The technical content and the code are the author's own.

This book shows you how to write nginx modules in Rust.

By the end, you will have a **statistics module that aggregates requests in shared memory and exposes them in Prometheus format**, complete with integration tests. It takes configuration directives, keeps counts across worker processes, and answers `/status` in a format Grafana can read. It is built to a standard you could put into production.

## Why Rust?

For a long time, nginx modules were written in C. Pointer lifetimes, freeing the memory you allocate, concurrent access to shared memory: the module author is responsible for all of it. Get any of it wrong and a worker process crashes, or keeps serving corrupted data. That is because C leaves memory management to the developer in the first place.

[ngx-rust](#) is the official Rust binding from F5/NGINX. Because it is Rust, the compiler takes over the part of memory management that can be checked mechanically, which C never does. Workers crashing out of nowhere or serving corrupted data become risks you rule out before they happen.

## Note

There is an official introduction: [Extending NGINX with Rust: An Alternative to C](#) announced ngx-rust, and it is usually the first search result.

It was, however, **written against 0.3.0-beta**, and the details differ from the main branch as of September 17, 2026, which this book targets. Keep the following in mind when comparing the two:

| The article                                             | The version this book targets                                                                               |
|---------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| The trait is called <code>HTTPModule</code>             | <code>HttpModule</code>                                                                                     |
| Two crates: <code>nginx-sys</code> and <code>ngx</code> | <b>Four</b> ( <code>nginx-src</code> and <code>examples</code> were added; see <a href="#">Appendix A</a> ) |

The big picture still holds. Only the names have changed.

Here is one example. nginx has several kinds of memory with different lifetimes: memory that goes away with the request, memory that lives until the next configuration reload, and shared memory that every worker can see. **Where a value lives shows up in its type**, so if you try to store a value that points into worker-local memory in a shared-memory table, Rust will not compile it. C will. It compiles, and then it breaks when another worker reads the value. [Chapter 11](#) covers this in detail.

In C, the answer was to tell yourself to be careful. In Rust, the same constraint shows up as a type right in front of you and holds you in place. That is the advantage of writing nginx modules in Rust, and it comes up again and again throughout this book.

Some ground remains that the compiler cannot cover: calls into nginx’s C functions, and anything that crosses process boundaries. Rust lets you mark such places explicitly as “a human is responsible here,” and ngx-rust is designed to gather them into as few places as possible. We first learn the basics of Rust’s memory management and types in [Chapter 4](#) and [Chapter 5](#), then cover this in [Chapter 6](#).

## The Approach of This Book

Every explanation is tied to **code that has actually been run**.

Chapter 3 starts with the code for a simple module that works. Where the explanations need more than that, they draw on ngx-rust’s [examples](#) and on [nginx-acme](#). We also port nginx’s own `ngx_http_stub_status_module` and `ngx_http_not_modified_filter_module` to Rust, and verify the latter against 14 conditional-request cases. Finally, we write a Rust version of [nginx-module-vts](#), which I maintain, and pass 13 integration test files with 300 assertions. Where an API was missing, I sent it upstream as a PR to ngx-rust.

The code in the text lives in per-chapter sample repositories, where CI builds and tests it. Nothing here fails when you type it in.

Links to external code are pinned to commits so the line numbers do not drift.

| Project                                  | Pinned commit           |
|------------------------------------------|-------------------------|
| ngx-rust (main as of September 17, 2026) | <a href="#">f5aacab</a> |
| nginx 1.30.x                             | <a href="#">017cf98</a> |
| nginx-acme                               | <a href="#">aabca6b</a> |
| ngx_http_stub_status_module              | <a href="#">8e80d8b</a> |

## What You Should Know

This book assumes **experience running nginx**. If you have written an `nginx.conf` and know how `server` and `location` relate and what `proxy_pass` does, you know enough. You do not need to have written a module in C.

It does not assume any Rust experience. This book is not a general Rust primer and does not try to cover the whole language. Instead, **it explains what you need to read ngx-rust code, using ngx-rust’s own code** (Part 1). Ownership and `unsafe` do not come up as abstract examples, but as “why this line in ngx-rust is written the way it is.”

You can read the book without **knowing C**, but it helps a great deal. If you have called C APIs from another language, even better. Some passages quote the nginx source to show how something is done in C; you can skip them and still follow the main thread.

# How the Book Is Organized

The book has three parts, preceded by a **getting-started section that gives you something that runs**.

## Getting Started (Chapters 1–3)

You set up an environment, then type in a minimal module and run it. By the end of [Chapter 3](#), an nginx you built yourself serves a response you wrote.

There is almost no explanation. The only goal is to **have something working first**. Every explanation after that comes back to these 100 lines.

## Part 1 — The Rust You Need to Read ngx-rust (Chapters 4–6)

Starting from the code you typed in, this part explains from the Rust side why it has the shape it does. You learn the minimum needed to understand that: ownership, `Option` and `Result`, traits and monomorphization, lifetimes, and `unsafe`. There are no toy examples; everything is tied to real ngx-rust code. Since you have already seen it work, none of it stays abstract.

If you have written FFI code in Rust, feel free to skip this part. [The opening page of Part 1](#) explains how.

## Part 2 — Writing Modules (Chapters 7–14)

This part goes through how to write each component: the three levels of configuration, the 11 phases, working with requests and responses, shared memory, filters, and load balancers.

The centerpiece is shared memory. nginx workers are processes, not threads, so even keeping a count across workers requires shared memory.

## Part 3 — Building a Real Module (Chapters 15–21)

With the tools from Parts 1 and 2, you build a statistics module. It aggregates in shared memory, exposes the results in Prometheus format, and has its correctness checked by integration tests.

Three more chapters cover what comes after: the pitfalls I actually ran into, what to do when `ngx-rust` is missing something you need, and a case study of the design decisions.

## Appendices (A–D)

There are four.

| Appendix          | Contents                                                             |
|-------------------|----------------------------------------------------------------------|
| <a href="#">A</a> | Where things live in the <code>ngx</code> crate, with a mapping to C |
| <a href="#">B</a> | Glossary                                                             |
| <a href="#">C</a> | How red-black trees work, by reading nginx’s C implementation        |
| <a href="#">D</a> | Trait index: look up any trait that appears in the text by name      |

A, B, and D are not meant to be read straight through; they are for **looking things up while you write code**. C is a reading for those who have not studied data structures, and you only need it if you get curious about what is inside shared memory in [Chapter 11](#).

## Terminology

Terms come from both nginx and Rust. Those used without explanation are collected in the glossary ([Appendix B](#)). If terms like “cycle,” “pool,” “FFI,” or “raw pointer” are unclear when they first appear, look there first.

## Tested Environments

|          |                                                           |
|----------|-----------------------------------------------------------|
| ngx-rust | main as of September 17, 2026 ( <a href="#">f5aacab</a> ) |
| nginx    | 1.30.4                                                    |
| Rust     | 1.85.0 or later (MSRV); tested on 1.90.x                  |
| OS       | Ubuntu 24.04 (x86_64) / macOS 14 (Apple Silicon)          |

In some places the page size directly affects behavior, so **everything was checked on both x86\_64 Linux (4 KiB pages) and Apple Silicon (16 KiB pages)**. [Chapter 19](#) explains why this matters.

# Chapter 2: Setting Up the Environment

By the end of this chapter, you will be able to build nginx from source and start it. We will not write a module yet; this is only the foundation.

Getting it right now pays off later: when a build fails in a later chapter, you will not have to wonder whether your code or your environment is to blame.

## What You Need

| Purpose               |                                                                                    |
|-----------------------|------------------------------------------------------------------------------------|
| Rust 1.85.0 or later  | ngx-rust's MSRV (Minimum Supported Rust Version). Install with <code>rustup</code> |
| A C compiler          | Building nginx itself. Either clang or gcc works                                   |
| <code>make</code>     | Same as above                                                                      |
| PCRE2, zlib, OpenSSL  | nginx's standard dependencies                                                      |
| <code>libclang</code> | Used by bindgen to read the nginx headers                                          |

## Ubuntu / Debian

```
sudo apt-get update
sudo apt-get install -y \
    build-essential curl git \
    libclang-dev libpcre2-dev zlib1g-dev libssl-dev

curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s -- -y
source "$HOME/.cargo/env"
```

## macOS

```
xcode-select --install
brew install pcre2 openssl@3

curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s -- -y
source "$HOME/.cargo/env"
```

`libclang` ships with the Command Line Tools, so there is nothing else to install.

## Getting the nginx Source

```
NGINX_VERSION=1.30.4

curl -fsSL "https://nginx.org/download/nginx-${NGINX_VERSION}.tar.gz" | tar xz
cd "nginx-${NGINX_VERSION}"
./configure --with-compat --with-debug
make -j"$(nproc 2>/dev/null || sysctl -n hw.ncpu)"
```

If `objs/nginx` exists, the build succeeded.

```
./objs/nginx -v
# nginx version: nginx/1.30.4
```

Both `configure` options are there for a reason.

`--with-compat` is required for dynamic modules. An nginx built without it rejects modules loaded with `load_module`, complaining that the signature does not match.

`--with-debug` enables debug logging. When writing a module, there will always come a point where you want to trace the flow with `error_log ... debug;`, so include it in development builds. Production builds do not need it.

## Match the Version Exactly

`--with-compat` does not mean nginx will load anything. `load_module` performs two checks:

```

/* src/core/nginx_module.c:170-182 */
if (module->version != nginx_version) {
    "module \"%V\" version %ui instead of %ui"
}
if (ngx_strcmp(module->signature, NGX_MODULE_SIGNATURE) != 0) {
    "module \"%V\" is not binary compatible"
}

```

→ [Source](#)

| Check     | What it compares                                                          | Does <code>--with-compat</code> help? |
|-----------|---------------------------------------------------------------------------|---------------------------------------|
| version   | The version number itself; 1030004 for 1.30.4                             | No                                    |
| signature | Pointer width, width of <code>time_t</code> , the set of enabled features | Yes                                   |

The first check requires an exact match down to the patch version. A module built against the 1.30.4 source will not load into nginx 1.30.3.

```
nginx: [emerg] module "/path/to/libngx_hello.so" version 1030004 instead of 1030003
```

`--with-compat` only affects the second check. It evens out ABI differences caused by which features are enabled; it does not bridge version differences.

### ⚠ Warning

This is where you will get stuck if you want to drop a module into an nginx you already have.

```
nginx -v          # check the version of the nginx you will run
```

Fetch **the source for exactly that version** and build against it. If the module is for your distribution's nginx, match that package's version. Whenever you upgrade nginx, rebuild the module.

The steps in this book link the module statically into an nginx you build yourself, so this problem does not come up. Because it does not come up, the first time you hit it is when you switch to a dynamic module.

# Three Ways ngx-rust Finds nginx

This is the first place people trip up. ngx-rust's build script reads the nginx headers and runs them through bindgen, so you have to tell it **which nginx to build against**.

| Method                                      | How                               | When to use it                             |
|---------------------------------------------|-----------------------------------|--------------------------------------------|
| Download and build the source automatically | <code>--features vendored</code>  | Quick experiments                          |
| Use an existing source tree                 | <code>NGINX_SOURCE_DIR=...</code> | Building for an nginx you built yourself   |
| Use an already-built <code>objs</code>      | <code>NGINX_BUILD_DIR=...</code>  | Same, when you know it has been configured |

If you specify nothing, you get this:

```
"nginx-sys/vendored" feature is disabled and neither
NGINX_SOURCE_DIR nor NGINX_BUILD_DIR is set
```

**This book uses** `NGINX_BUILD_DIR`, for two reasons. First, `vendored` makes the crate build its own private nginx, which is not the `objs/nginx` you just built. Load the result as a dynamic module and the version check rejects it.

```
nginx: [emerg] module "... " version 1028003 instead of 1030004
```

Second, `NGINX_SOURCE_DIR` requires a **configured tree**. Pass it a source tree that has only been extracted, and the `objs/nginx_auto_config.h` it expects to find is missing, so the build fails like this:

```
thread 'main' panicked at nginx-sys/build/main.rs:
  Invalid nginx build directory: ".../nginx-1.30.4/objs". No such file or directory
```

`NGINX_BUILD_DIR` points directly at `objs`, so this mix-up cannot happen.

```
export NGINX_BUILD_DIR="$PWD/objs" # points to nginx-1.30.4/objs
```

## Note

Watch the order of steps when you write CI. You end up running nginx's configure twice: building the module requires a configured `objs/`, and building nginx requires the built module. Run the first configure to create `objs/`, build the module, then configure again with `--add-module` and run `make`.

## ngx\_feature — nginx's Build Configuration as Rust `cfg`

There is one more mechanism that will save you trouble later if you know about it now.

nginx gains and loses features depending on `configure` options. Without `--with-http_ssl_module`, the SSL-related struct fields simply do not exist. ngx-rust reads this from `objs/nginx_auto_config.h` and **exposes it as Rust `cfg`**.

```
[cfg(ngx_feature = "http_ssl")]
fn something_using_ssl() { /* ... */ }
```

This is not the same as a Cargo feature. You choose Cargo features yourself in `Cargo.toml`; `ngx_feature` is **decided for you by how your nginx was built**.

Here is an example where it matters. The connection counters used in Part 3 look like this on the nginx side:

```
/* src/event/nginx_event.h */
#if (NGX_STAT_STUB)
extern ngx_atomic_t *ngx_stat_active;
/* ... */
#endif
```

In an nginx built without `--with-http_stub_status_module`, this variable does not exist. So the Rust side must wrap it in `[cfg(ngx_feature = "stat_stub")]` too; if you do not, you get a link error.

The nasty part goes the other way: **code removed by `cfg` disappears silently**. The tests disappear with it, so you see `0 tests` and a green result. The tests did not pass; they never existed.

```
# Enable it explicitly to test code that needs stat_stub
NGX_CONFIGURE_ARGS="--with-http_stub_status_module" \
  cargo test --features vendored
```

## Checking That It Works

Let's start nginx with a minimal configuration.

```
mkdir -p /tmp/nginxtest/{conf,logs,html}
cat > /tmp/nginxtest/conf/nginx.conf <<'EOF'
daemon off;
master_process off;
events { worker_connections 64; }
http {
    server {
        listen 127.0.0.1:8080;
        location / { return 200 "ok\n"; }
    }
}
EOF

./objs/nginx -p /tmp/nginxtest/ -c /tmp/nginxtest/conf/nginx.conf &
curl http://127.0.0.1:8080/
# ok
```

`daemon off;` `master_process off;` are debugging settings. nginx runs in the foreground without forking workers into separate processes, so `ctrl-c` stops it and attaching a debugger is easier.

## Integration Testing Tools

In Part 3 we write integration tests for the module. They use the same Perl framework as nginx itself, so install it now.

```
cpanm --notest Test::Nginx
```

If you do not have `cpanm`, install it with `cpan App::cpanminus`. [Chapter 18](#) covers how to use it.

---

The foundation is in place. In the next chapter, we write a module that actually works.

# Chapter 3: Running Your First Module

In this chapter we write a module. A location containing a `hello;` directive will return a response built in Rust.

Explanations come later. We get it running first, then take it apart in [Chapter 7](#). Things sink in far better when you already have something working in front of you.

## What We Are Building

```
location /hello {  
    hello;  
}
```

```
$ curl http://127.0.0.1:8090/hello  
Hello from Rust
```

## Creating the Project

```
cargo new --lib ngx_hello  
cd ngx_hello
```

Make `Cargo.toml` look like this:

```
[package]  
name = "ngx_hello"  
version = "0.1.0"  
edition = "2024"  
  
[lib]  
crate-type = ["cdylib"]  
  
[dependencies]  
ngx = { git = "https://github.com/nginx/nginx-rust", rev =  
    "f5aacab54161d6c3602a9f135d4ac1c702098452" }
```

`crate-type = ["cdylib"]` is the key line. It makes Cargo produce output that nginx can link as a C library. Without it you only get an `rlib` for Rust, which nginx cannot see.

The dependency is pinned to a commit. ngx-rust's API is still changing, and pinning keeps the book and your local behavior in sync. For real-world use, specifying a version such as `ngx = "0.5"` is fine.

## The config File for Building into nginx

nginx's `--add-module` expects a shell script named `config` in that directory. For a Rust module, its job is to **have nginx link the already-built shared library**.

Put `config` at the project root:

```
ngx_addon_name=ngx_http_hello_module

# cargo produces .dylib on macOS and .so on Linux
if test -f "$ngx_addon_dir/target/release/libngx_hello.dylib"; then
    ngx_hello_lib="$ngx_addon_dir/target/release/libngx_hello.dylib"
else
    ngx_hello_lib="$ngx_addon_dir/target/release/libngx_hello.so"
fi

if test -n "$ngx_module_link"; then
    ngx_module_type=HTTP
    ngx_module_name=ngx_http_hello_module
    ngx_module_libs="$ngx_hello_lib"
    . auto/module
else
    HTTP_MODULES="$HTTP_MODULES ngx_http_hello_module"
    NGX_ADDON_DEPS="$NGX_ADDON_DEPS $ngx_hello_lib"
    CORE_LIBS="$CORE_LIBS $ngx_hello_lib"
fi
```

The `if test -n "$ngx_module_link"` branch switches between the style used since nginx 1.9.11 and the older one. In practice only the first branch runs, but writing both is the convention.

The name in `ngx_module_name` **must match** the name of the `static` we define on the Rust side next. nginx writes this name into the `ngx_modules.c` it generates and resolves it at link time.

## The Module Itself

Replace `src/lib.rs` entirely:

```

use core::ffi::{c_char, c_void};
use core::ptr;

use ngx::core::Status;
use ngx::ffi::{
    ngx_command_t, ngx_conf_t, ngx_http_module_t, ngx_module_t, ngx_str_t, ngx_uint_t,
    NGX_CONF_NOARGS, NGX_HTTP_LOC_CONF, NGX_HTTP_LOC_CONF_OFFSET, NGX_HTTP_MODULE,
};
use ngx::http::{self, HttpModuleLocationConf, NgxHttpCoreModule};
use ngx::{http_request_handler, ngx_string};

// ---- Directive table -----

static mut NGX_HTTP_HELLO_COMMANDS: [ngx_command_t; 2] = [
    ngx_command_t {
        name: ngx_string!("hello"),
        type_: (NGX_HTTP_LOC_CONF | NGX_CONF_NOARGS) as ngx_uint_t,
        set: Some(hello_directive),
        conf: NGX_HTTP_LOC_CONF_OFFSET,
        offset: 0,
        post: ptr::null_mut(),
    },
    ngx_command_t::empty(), // terminator; without it nginx reads past the table and
                             crashes
];

// ---- Module context -----

static NGX_HTTP_HELLO_MODULE_CTX: ngx_http_module_t = ngx_http_module_t {
    preconfiguration: None,
    postconfiguration: None,
    create_main_conf: None,
    init_main_conf: None,
    create_srv_conf: None,
    merge_srv_conf: None,
    create_loc_conf: None,
    merge_loc_conf: None,
};

// ---- Module definition -----

#[used]
#[allow(non_upper_case_globals)]
#[unsafe(no_mangle)]
pub static mut ngx_http_hello_module: ngx_module_t = ngx_module_t {
    ctx: &raw const NGX_HTTP_HELLO_MODULE_CTX as _,
    commands: unsafe { &raw mut NGX_HTTP_HELLO_COMMANDS[0] },
    type_: NGX_HTTP_MODULE as ngx_uint_t,
    ..ngx_module_t::default()
};

// ---- Handler -----

const CONTENT_TYPE: ngx_str_t = ngx_string!("text/plain");

http_request_handler!(hello_handler, |request: &mut http::Request| {
    let body = "Hello from Rust\n";

    // We do not read the request body, so discard it
    let rc = request.discard_request_body();

```

```

    if rc != Status::NGX_OK {
        return rc;
    }

    // Allocate the buffer from the request pool; it goes away once the response is
sent
    let Some(mut buf) = request.pool().create_buffer_from_str(body) else {
        return http::HTTPStatus::INTERNAL_SERVER_ERROR.into();
    };

    request.set_status(http::HTTPStatus::OK);
    request.set_content_length_n(buf.len());

    // Content-Type has no accessor, so write to the raw struct
    let raw = request.as_mut();
    raw.headers_out.content_type = CONTENT_TYPE;
    raw.headers_out.content_type_len = CONTENT_TYPE.len();
    raw.headers_out.content_type_lowercase = ptr::null_mut();

    use ngx::core::Buffer;
    buf.set_last_buf(request.is_main());
    buf.set_last_in_chain(true);

    let rc = request.send_header();
    if rc == Status::NGX_ERROR || rc > Status::NGX_OK || request.header_only() {
        return rc;
    }

    let mut out = ngx::ffi::ngx_chain_t {
        buf: buf.as_ngx_buf_mut(),
        next: ptr::null_mut(),
    };
    request.output_filter(&mut out)
});

// ---- The directive's set handler -----

extern "C" fn hello_directive(
    cf: *mut ngx_conf_t,
    _cmd: *mut ngx_command_t,
    _conf: *mut c_void,
) -> *mut c_char {
    // SAFETY: nginx always passes a valid cf to configuration handlers
    let cf = unsafe { cf.as_mut().unwrap() };

    let Some(clcf) = NgxHttpCoreModule::location_conf_mut(cf) else {
        return ngx::core::NGX_CONF_ERROR;
    };
    clcf.handler = Some(hello_handler);

    ngx::core::NGX_CONF_OK
}

```

That is a little over 100 lines. It is fine if you cannot read it yet. [Part 1](#) and [Chapter 7](#) take it apart piece by piece.

For now, the only thing to take away is that **these three pieces together make a module**:

1. **The directive table** ( `NGX_HTTP_HELLO_COMMANDS` ): which words you add to the configuration file
2. **The module definition** ( `ngx_http_hello_module` ): the entry point nginx recognizes as a module
3. **The handler** ( `hello_handler` ): the function that actually builds the response

## Building

First, build the module as Rust. Point it at the `objs` of the nginx you prepared in [Chapter 2](#).

```
export NGINX_BUILD_DIR=/path/to/nginx-1.30.4/objs
cargo build --release
```

This produces `target/release/libngx_hello.dylib` ( `.so` on Linux).

Next, rebuild nginx so that it pulls in that library.

```
cd /path/to/nginx-1.30.4
auto/configure --with-compat --with-debug --add-module=/path/to/ngx_hello
make -j4
```

### Note

The build steps must run in a fixed order. Building the module needs the nginx headers (a configured `objs/`), and linking nginx needs the built module. Since each needs the other, you go around like this:

1. Run nginx's `configure` once (this creates `objs/`)
2. `cargo build --release` (this builds the module)
3. Run nginx's `configure` again with `--add-module`, then `make`

This is what [Chapter 2](#) meant by “you run nginx’s configure twice.”

# Running It

```
mkdir -p /tmp/hello/{conf,logs,html}
cat > /tmp/hello/conf/nginx.conf <<'EOF'
daemon off;
master_process off;
events { worker_connections 64; }
http {
    server {
        listen 127.0.0.1:8090;
        location /hello { hello; }
        location /      { return 200 "not hello\n"; }
    }
}
EOF

./objs/nginx -p /tmp/hello/ -c /tmp/hello/conf/nginx.conf
```

Send a request from another terminal.

```
$ curl -i http://127.0.0.1:8090/hello
HTTP/1.1 200 OK
Server: nginx/1.30.4
Date: Tue, 01 Sep 2026 13:37:05 GMT
Content-Type: text/plain
Content-Length: 16
Connection: keep-alive

Hello from Rust

$ curl http://127.0.0.1:8090/
not hello
```

Check that `Content-Type` and `Content-Length` are set as intended, and that locations without `hello;` are unaffected.

## Troubleshooting

**unknown directive "hello"** The module is not in nginx. Check that you re-ran `configure` with `--add-module` and that `make` actually ran. If `nginx -V` shows the path under `configure arguments`, the module is in.

**Link error: \_ngx\_http\_hello\_module not found** `ngx_module_name` in `config` and the name of the `static` on the Rust side do not match. Also check that `# [unsafe(no_mangle)]` is there. Without it, Rust mangles the name and C cannot find it.

## **cargo build cannot find nginx**

```
"nginx-sys/vendored" feature is disabled and neither
NGINX_SOURCE_DIR nor NGINX_BUILD_DIR is set
```

`NGINX_BUILD_DIR` is not being passed. Check that you ran `cargo` in the same shell where you ran `export`.

**Invalid nginx build directory ... No such file or directory** The directory you pointed to has no `objs/nginx_auto_config.h`. nginx has never been configured.

**module "... " version 1030004 instead of 1030003** The nginx source you built the module against and the nginx you are loading it into are different versions. They must match down to the patch version. Check with `nginx -v` and rebuild against the same source ([Chapter 2](#)).

**module "... " is not binary compatible** The versions match, but the `configure` options differ. **Both the module and nginx need** `--with-compat` ([Chapter 2](#)).

**HEAD requests return a body** This happens on aarch64, such as Apple Silicon, when `ngx` was built with `bindgen` 0.72.x. Older releases, including 0.5.0, are affected.

`request.header_only()` reads the wrong value, so the check does nothing. The `f5aacab` this book pins uses `bindgen` 0.73.2 and is not affected, but changing `rev` to an older version brings it back (see [Chapter 19](#), “`bindgen` computes bitfield positions on its own”). You will not notice with `curl -I`, so it is easy to miss entirely.

---

You now have something that works. From here, we read these 100 lines from two directions.

Part 1 reads them from the Rust side. What is `let ... else`? What does `unsafe` guarantee? Why `static mut`? It explains why the code you wrote has the shape it does.

Part 2 reads them from the nginx side. How is the directive table read? When is the handler called? How do you receive the three levels of configuration? From there it moves on to the pieces hello did not have: phases, shared memory, and filters.

If you have written FFI code in Rust, skip Part 1 and start with Part 2.

 **Tip**

This is the end of the free sample. The rest of the book, from Part 1 onward, is described on [Get the Full Book](#).

# Part 1 — The Rust You Need to Read ngx-rust

The 100 lines you wrote in Getting Started used several constructs whose explanation we put off.

```
let Some(mut buf) = request.pool().create_buffer_from_str(body) else { ... };
let cf = unsafe { cf.as_mut().unwrap() };
#[used]
#[unsafe(no_mangle)]
pub static mut ngx_http_hello_module: ngx_module_t = ...
```

What is `let ... else`? What does `unsafe` mean, and what does it not mean? Why `static mut`? This part answers those questions.

## The Approach of This Part

This is not a Rust primer. Instead of covering the whole language, **it explains what you need to read ngx-rust code, using ngx-rust's own code.**

None of the examples are made up. Every one comes from ngx-rust itself or from a module that actually runs, with a link to the exact lines. We do not take the detour of explaining ownership with cats and dogs before coming back to ngx-rust.

The trade-off is that some topics a general primer covers never come up: multithreading, trait objects, defining macros, designing error types. When you need them, turn to [The Rust Programming Language](#).

## What We Use as Material

We start by going back to the hello module you typed in during [Chapter 3](#). Those 100 lines already contain ownership, borrowing, `unsafe`, and traits. Since you have already worked with those lines yourself, each explanation can start from “why was it written this way?”

**hello alone is not enough**, though. At 100 lines, it lacks things like these:

| Missing from hello                                   | Where we find it instead             |
|------------------------------------------------------|--------------------------------------|
| Moving a value somewhere else for safekeeping        | <code>examples/shared_dict.rs</code> |
| Locks that span processes                            | Same                                 |
| <code>unsafe</code> for calling C functions directly | Same                                 |
| <code>unsafe impl</code>                             | <code>src/core/slab.rs</code>        |

For the gaps, we mainly use ngx-rust's `examples/shared_dict.rs`. It is a 425-line module that covers configuration directives, shared memory, an rbtree, and variable handlers. It ships with ngx-rust and it runs, so you can build it locally and check things for yourself.

Each section says which of the two it draws on. Anything hello can explain is explained with hello; we only step outside it for what hello cannot show.

## Chapters in This Part

| Chapter                                                                               | What it covers                                                                        |
|---------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| <a href="#">Chapter 4. Ownership and Borrowing</a>                                    | Who cleans up a value, and what happens when you hand a value over to the C world     |
| <a href="#">Chapter 5. Expressing It in Types — Option, Result, Traits, Lifetimes</a> | Replacing null and error codes, and how monomorphization pays off at the FFI boundary |
| <a href="#">Chapter 6. <code>unsafe</code></a>                                        | What it means and what it does not, and how to write <code>SAFETY</code> comments     |

## Who Can Skip This Part

If you have written FFI code in Rust, you can skip this part. Start with Part 2 and come back if you get stuck.

Two sections are still worth a look, though. They are where experience from other languages does not carry over directly.

- **“Monomorphization Pays Off at the FFI Boundary” in Chapter 5:** this is the very mechanism ngx-rust uses to make C callbacks look type-safe. It is where you find out why `add_phase_handler::()` works.
- **The `unsafe` sections of Chapter 6:** they explain that `unsafe` does not mean “dangerous” but is a declaration that a human guarantees what the compiler cannot. All of ngx-rust’s code is written on that premise.

If you are new to Rust, read the chapters in order. By the end, you will be able to explain every line of the code you typed in during Getting Started.

# Get the Full Book

Chapters 1–3 and the opening page of Part 1 are free to read here. The full book continues with:

- **Part 1 — The Rust You Need to Read ngx-rust:** ownership, `Option` and `Result`, traits, lifetimes, and `unsafe`, all explained through real ngx-rust code
- **Part 2 — Writing Modules:** configuration, phases, requests and responses, allocators, shared memory and inter-process locking, filters, and load balancers
- **Part 3 — Building a Real Module:** a statistics module that aggregates in shared memory, exposes Prometheus metrics, and is checked by integration tests, plus pitfalls, contributing to ngx-rust, and a case study
- **Appendices:** an API map of the `ngx` crate, a glossary, red-black trees, and a trait index

The full book is available on Leanpub as PDF and EPUB: [Writing nginx Modules in Rust](#). Buying it there also gets you every future update.

The complete Japanese edition is available on Zenn: [Rustで書くnginxモジュール](#).