Nidus: Explicit Application Composition for Rust Backend Services

Victor Bona
Independent Researcher, Blumenau, Brazil
victor.bona@hotmail.com

September 2026

Abstract

A backend application needs a composition model as well as a request handler. Its components must declare dependencies, initialize shared resources, expose routes, and coordinate operational behavior. Nidus organizes these responsibilities through explicit modules, typed providers, and controller metadata, while retaining Axum routing, Tower middleware, and Tokio execution. This paper explains the design through three separate stages: generated Rust declarations, application construction, and request execution. It describes the guarantees provided at each stage, follows a small feature from module declaration to handler invocation, and identifies the costs and limits of the abstraction. The implementation uses a runtime container keyed by Rust types and validates module structure during bootstrap. Module exports describe composition contracts; they do not enforce runtime access isolation. The account is anchored to Nidus 1.0.17 and supported by source inspection and focused executable checks.

1 Problem and design

An HTTP route describes request handling. A growing backend also needs to determine which component provides a database pool, which features consume it, when initialization occurs, and how resources are released. These composition decisions exist even when they are distributed across constructors and entrypoint code.

Nidus makes that composition explicit. A root module identifies the application. Imported modules declare its features and infrastructure. Providers construct and share typed values. Controllers expose feature behavior through HTTP routes. The resulting structure can be checked before the server accepts traffic and inspected separately from request execution. The intended benefit is a common application vocabulary across features, with construction rules that remain visible in source.

The design draws on NestJS's modules, providers, controllers, imports, and exports [1, 2]. Nidus expresses them through Rust types, macros, and factories. Axum supplies HTTP routing and extraction; Tower supplies composable service and layer abstractions [3, 4].

This paper describes the committed 1.0.17 implementation at `99642ed0018c` [1]. Its scope is architectural: what the framework organizes, where it performs work, and which responsibilities remain with the application. It claims neither a new dependency-injection algorithm nor a measured productivity improvement.

2 The application model

A Nidus module records four groups: imports, providers, controllers, and exports. Imports name other modules. Providers identify values to register. Controllers identify route-producing components. Exports identify local providers offered through the module's declared interface.

The module graph is directed. An edge from a feature to an infrastructure module means that the feature imports that module. Typed imports carry a definition factory, allowing discovery to proceed recursively from the root. String-only imports require the corresponding definitions to be supplied explicitly. This distinction separates executable discovery from descriptive names.

At bootstrap, the graph rejects missing imports, import cycles, duplicate declarations, exports that are not local providers, and conflicting or ambiguous provider names in declared visibility. Imported modules precede their importers during provider registration and initialization. The public inspection iterator uses name order, so display order and execution order are distinct.

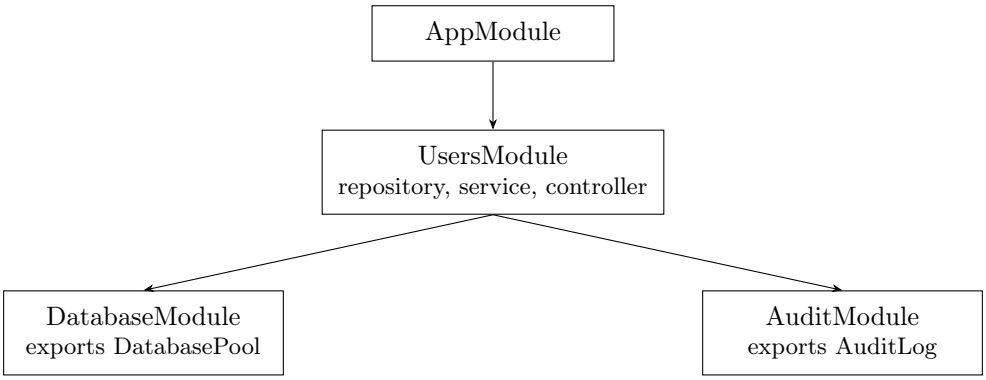

Figure 1: A module graph for one feature. Arrows mean imports. DatabaseModule and AuditModule register before UsersModule; UsersModule registers before AppModule. The graph describes composition, not HTTP request flow.

Module metadata and provider identity are different mechanisms. The graph validates names stored in module definitions. The runtime container uses `TypeId` keys and returns typed values through operations such as `resolve::<T>()`. Registration and resolution therefore retain Rust type information even though module diagnostics use names.

The container is shared across the application. A resolution request does not carry the identity of the requesting module. Exports consequently support validation of the declared architecture, but do not create private runtime containers or prevent direct resolution of a registered type. Rust visibility, crate boundaries, and application interfaces remain necessary when access must be restricted. Likewise, a valid module graph does not prove that every provider factory can resolve every dependency it will request.

3 Construction and execution

Nidus separates work across compilation, application construction, and request execution. This separation is the main technical constraint behind its ergonomic API.

3.1 Generated declarations

Procedural macros validate supported syntax and generate Rust implementations for module definitions, provider registration, controller construction, and route metadata. Rust then type-checks the generated program. An injectable field such as `Inject<UsersRepository>` becomes a typed resolution operation in the generated factory.

This is typed runtime dependency injection. Macro validation catches malformed declarations, and Rust checks type and trait requirements. Missing registrations, factory failures, and circular factory resolution remain runtime errors. Calling the mechanism compile-time dependency injection would overstate its guarantees.

3.2 Application construction

The facade call `Nidus::create::<AppModule>().build().await` discovers and validates the module graph, runs synchronous provider registrars, then runs asynchronous initializers. Both provider phases follow dependency order. Every registrar completes before asynchronous initialization begins. Initializers run sequentially with exclusive access to the container, and an error stops construction.

The builder then constructs module controllers, resolves their dependencies, and merges their routers. It checks repeated method/path pairs in controller metadata and composes configured OpenAPI, observability, and dashboard surfaces. Manually supplied Axum routers are merged through the same builder, but their arbitrary contents are not recovered as Nidus controller metadata.

The lower-level synchronous `Nidus::bootstrap` entrypoint validates the graph and registers providers. It does not execute asynchronous initializers or assemble module HTTP routes. Choosing an entrypoint therefore selects concrete construction behavior, not merely a different spelling of the same operation.

3.3 Provider lifetimes and requests

A provider factory specifies how to obtain a value. Its lifetime specifies when construction is repeated. Table 1 gives the three container lifetimes.

Lifetime	Resolution behavior
Singleton	One successfully constructed instance is cached and shared through `Arc<T>`. Factory-backed singletons are lazy unless explicitly resolved.
Transient	Each resolution constructs a fresh value. A consumer that retains that value also determines how long it remains alive.
Request	One instance is reused within an explicit request scope. Resolving it through the root container returns `RequestScopeRequired`.

Table 1: Construction frequency is determined by resolution. Transient does not mean one instance per HTTP request.

Module controllers are constructed while the router is built. Their generated handlers retain shared controller references and clone those references for requests. A service injected into such a controller is therefore resolved during construction. If that service is transient, the controller still retains that particular instance. Request-specific state requires an explicit scope, such as the

HTTP scope layer and `RequestScoped<T>` extractor. Nested request providers use a scope-aware factory to resolve dependencies from the same scope.

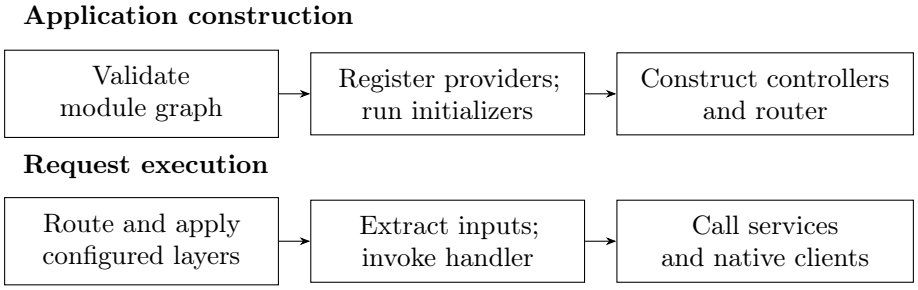

Figure 2: Two execution stages for a controller with injected shared state. Router construction establishes the retained references. Ordinary requests do not repeat module graph validation. Explicit request-scoped resolution adds work to the lower path.

The default controller path does not rediscover modules or traverse their import graph per request. It still incurs the costs of shared references, extraction, configured middleware, and application code. Explicit container resolution adds lookup and provider-lifetime costs wherever the application invokes it.

4 A feature from declaration to request

Consider the UsersModule in Figure 1. The following declaration assumes that the named types and imported modules are defined. It records the feature boundary without embedding connection setup or query logic:

```
#[module]
struct UsersModule {
    imports: (DatabaseModule, AuditModule),
    providers: (UsersRepository, UsersService),
    controllers: (UsersController,),
    exports: (UsersService,),
}
```

DatabaseModule registers a database pool through application initialization or an adapter. AuditModule registers an audit service. UsersRepository receives the pool; UsersService receives the repository and audit service. The controller receives UsersService. Each dependency is expressed through a typed field or an explicit factory.

```
#[controller("/users")]
struct UsersController {
    service: Inject<UsersService>,
}

#[routes]
impl UsersController {
    #[get("/:id")]
    async fn profile(&self, Path(id): Path<i64>) -> String {
        self.service.profile(id).email
    }
}
```

During construction, the provider factories establish the reference chain and the controller contributes its route. For `GET /users/42`, Axum extracts the identifier, the generated handler invokes `profile`, and the service calls its retained dependencies. The route declaration uses Nidus's path syntax, which the HTTP layer normalizes for Axum.

This excerpt is adapted from the repository's modular-monolith example; its repository returns an illustrative profile rather than issuing a database query. The separate realworld-api example supplies the larger composition with SQLite, validation, guards, and operational endpoints [5]. The distinction matters: the small example explains ownership and resolution, while the larger example exercises concrete integrations.

5 Operational boundaries

The same preference for explicit composition applies to lifecycle, middleware, and adapters. A capability only governs the operations that actually pass through it.

5.1 Lifecycle and HTTP shutdown

`LifecycleRunner` executes registered startup hooks in order. If a hook returns an error, it invokes shutdown on the previously successful hooks in reverse order and preserves rollback failures alongside the startup error. Normal shutdown attempts every registered hook in reverse order even when an earlier shutdown returns an error.

These are guarantees of the explicitly invoked runner. In 1.0.17, the facade builder initializes providers but does not attach a populated lifecycle runner. Lifecycle-aware core bootstrap is a separate entrypoint. HTTP graceful-shutdown methods accept a caller-supplied signal and delegate connection draining to Axum; they do not automatically invoke application lifecycle shutdown. Plain `listen` installs no shutdown signal. Applications must connect initialization, serving, draining, and resource cleanup deliberately. Hook rollback also does not imply transactional rollback of arbitrary provider-initializer side effects.

5.2 Request policy and introspection

Production API defaults are opt-in router and middleware composition: request identification, limits, timeouts, error envelopes, health checks, and other policies must be configured for the application. Module-composed guard declarations resolve a guard and run its check before invoking the controller method. Direct controller-to-router composition requires explicit guard

wiring. JSON validation can run through `ValidatedJson<T>`; recording metadata alone does not validate a payload.

OpenAPI and dashboard projections describe framework-owned metadata and observed operations. They do not recover all behavior of arbitrary Rust handlers or manually supplied routers. Similarly, `cargo nidus graph` analyzes source declarations rather than executing the deployed application's composition. These views are useful inspection surfaces within their respective coverage.

5.3 Native adapters

Data, broker, and telemetry adapters are separate installable crates. They expose native ecosystem clients and add integration with typed registration, lifecycle, health, or telemetry where supported. The shared integration layer standardizes envelopes and correlation metadata without imposing a universal queue interface [6].

This preserves backend semantics. Kafka offsets, RabbitMQ acknowledgments, and SQS visibility receipts remain backend-specific responsibilities. Durable jobs provide at-least-once delivery, so handlers must account for repeated execution. Adapter-owned admission and draining apply to operations routed through the adapter. Calls through exposed native clients remain outside that accounting unless the application explicitly coordinates them.

6 Evidence and cost model

The architectural account is supported by the pinned source, focused tests, and existing benchmark artifacts. The accompanying evidence manifest maps the paper's principal claims to implementation and test files. Checks run against an isolated snapshot, leaving the development checkout unchanged. The verification script records the exact commands, toolchain, results, and source hashes.

The focused run passed 75 tests covering module validation, registration order, asynchronous initialization, provider reuse and scope, lifecycle error handling, facade route construction, and the modular-monolith example. These checks establish the behavior of those cases. They do not constitute a full release qualification or prove correctness for all applications.

`nidus-testing` drives Axum services in memory without starting a TCP listener. At this source revision, its module bootstrap helpers call core bootstrap for validation and then create a separate test builder; they do not preserve the production application's full composition. Tests that need production router behavior must construct and exercise that router explicitly. Provider overrides also cannot retroactively replace values already captured by a controller.

Performance should be evaluated at the boundary where cost is introduced. Module validation and controller construction affect startup. Typed resolution includes a type-keyed lookup, shared-value handling, and any factory or scope work. Generated controller handlers retain shared references. Request middleware adds its own per-request work. Native I/O and business logic remain workload costs. Moving composition out of the ordinary request path does not establish zero overhead.

The repository separates dependency-resolution, route-construction, and request-lifecycle microbenchmarks, including raw Axum baselines where appropriate [7]. It also retains a qualified 1.0.12 end-to-end campaign with raw result summaries. That campaign used paced workloads, predates this paper's source revision, and does not include its complete harness in the repository.

It is historical evidence, not a current maximum-throughput result. No new latency, throughput, or capacity claim is made here.

7 Tradeoffs and scope

Nidus is useful when a backend benefits from a common declaration of feature boundaries, provider construction, and operational integration. The additional vocabulary has a cost: macros generate code to inspect, the container introduces runtime resolution, and composition errors can appear after compilation. For a small service with a few dependencies, direct constructors and an Axum router may already express the whole application clearly.

The 1.0.17 boundaries also identify concrete directions for improvement: stronger agreement between production and test composition, lifecycle ownership across serving and cleanup, and inspection derived from the same application representation used at runtime. These are directions beyond the implementation described here, not prerequisites silently assumed by the paper.

Nidus's central design is an explicit composition layer for Rust backend services. Modules describe how features fit together; providers construct typed values; controllers expose behavior through the existing HTTP ecosystem. Its usefulness depends on making those relationships easier to understand while keeping their execution and operational limits visible.

Acknowledgments

This paper was developed in an AI-assisted workflow directed by the author, using Codex (OpenAI). Its technical account is tied to the source revision and executable checks recorded in the accompanying artifact.

References

[1] V. Bona. *Nidus*, version 1.0.17, source revision `99642ed0018c`. Pinned source tree. See `nidus-core`, `nidus-macros`, `nidus-http`, and the facade's `app.rs`.

[2] NestJS. *Modules*. Official documentation. `https://docs.nestjs.com/modules`. Accessed September 2026.

[3] Tokio contributors. *Axum 0.8.9*. Crate documentation. `https://docs.rs/axum/0.8.9/axum/`.

[4] Tower contributors. *Tower 0.5.3*. Crate documentation. `https://docs.rs/tower/0.5.3/tower/`.

[5] Nidus. *Modular-monolith and realworld-api examples*. Source revision as in [1]. Example sources and tests.

[6] Nidus. *First-party integrations*. Source revision as in [1]. Adapter contracts, delivery semantics, and lifecycle boundaries.

[7] Nidus. *Performance and published benchmark data*. Source revision as in [1]. Methods and recorded results; Historical result artifacts.