Downloaded from buildwithruud.com · Kami Document System

Multi-Tenant Rust Architecture at the Edge: Zero-Trust Is

2026.11.21 · /blog/multi-tenant-rust-architecture-atlas

1. The Multi-Tenant Boundary Problem

Every multi-tenant platform faces a fundamental architectural tension: **operational density versus blast-radius isolation**. Running a dedicated compute cluster and database per tenant maximizes isolation at the cost of linear operational overhead and idle hardware. Conversely, collapsing tenants into a single shared runtime without compile-time boundary enforcement invites the most catastrophic class of distributed defect — cross-tenant data bleed.

When we architected **Atlas** (the cloud control plane powering Folio, Anchor, Plaza, and Beacon), the invariant was non-negotiable: a single bare-metal K3s and NixOS footprint had to serve arbitrary custom domains and product verticals while guaranteeing that no request handler, background worker, or AI agent tool call could ever read or mutate a row outside its resolved tenant scope.

2. Edge Resolution & Capability Injection

In a conventional web service, tenancy is often passed as a request path parameter or parsed inside individual controller functions. That pattern fails because security depends on every engineer remembering to add a `WHERE tenant_id = $1` clause in every query.

In Atlas, tenant resolution happens once at the HTTP/TLS ingress boundary before route dispatch:

1. **SNI & Host Extraction**: The edge middleware inspects the incoming `Host` header (or `X-Forwarded-Host` from the ingress proxy) and resolves it against an in-memory, read-through cache backed by `app_domains` and `app_instances`.
2. **Immutable Extension Injection**: The resolved `TenantContext(Option<Uuid>)` and `AppInstanceContext` are injected as typed request extensions.
3. **Fail-Closed Server Functions**: Every server function and REST handler extracts `Extension<TenantContext>` at the top of its execution frame. Unknown hosts fail closed before touching the connection pool.

rust

```
[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub struct TenantContext(pub Option<uuid::Uuid>);
pub async fn resolve_tenant_middleware(
State(state): State<AppState>,
mut req: Request<Body>,
next: Next,
) -> Result<Response, StatusCode> {
let host = extract_normalized_host(req.headers()).ok_or(StatusCode::BAD_REQUEST)?;
let tenant_id = state.domain_resolver.resolve_host(&host).await?;
req.extensions_mut().insert(TenantContext(Some(tenant_id)));
Ok(next.run(req).await)
}
```

3. Configuration-Driven UI Grammar & Isomorphic Rust SSR