Toggly Rust

Local evaluation, owned context, native server integrations

Install

```
[dependencies]
toggly = "0.6"
toggly-macros = "0.6" # optional
serde_json = "1"
tokio = { version = "1",
  features = ["macros", "rt-multi-thread"] }
```

Use the adapter guide for its host dependencies. Set TOGGLY_APP_KEY locally; never commit it.

Refresh and diagnostics

```
use toggly::TogglyClient;

pub async fn refresh(client: &TogglyClient)
  → toggly::Result<()>
{
  client.refresh().await?;
  println!("ETag: {:?}", client.etag().await);
  println!("Error: {:?}",
           client.last_error().await);
  Ok(())
}
```

Decision TTL defaults to 60s; polling defaults to 300s. clear_cache().await also clears in-memory definitions/JWKS. Refresh before evaluating loaded flags again. There is no client snapshot export or public cache-statistics accessor.

Initialize and close

```
use toggly::{EvalContext, TogglyClient};

#[tokio::main]
async fn main()
  → Result<(), Box<dyn std::error::Error>>
{
  let context = EvalContext::builder()
    .identity("alice").group("beta").build();
  let client = TogglyClient::builder()
    .app_key(std::env::var("TOGGLY_APP_KEY")?)
    .environment("Production")
    .build().await?;
  let result = client.is_enabled(
    "new-dashboard", context).await;
  client.close().await;
  println!("Enabled: {}", result?);
  Ok(())
}
```

Startup downloads shared app/environment definitions. Evaluation uses the supplied context locally; there is no separate startup identity call.

User and request context

```
use toggly::{EvalContext, HttpRequestMapper};

pub fn context(user_id: &str) → EvalContext {
  EvalContext::builder()
    .identity(user_id)
    .groups(vec!["beta".into()])
    .claim("role", "admin")
    .trait_value("plan", "enterprise")
    .request(HttpRequestMapper::from_http_headers([
      ("accept-language", "en-US"),
      ("cf-ipcountry", "US"),
    ]))
    .build()
}
```

Replace illustrative claims/groups with verified auth data. Trust country headers only from your proxy. Traits do not automatically become claims.

Multi-key gate

```
use toggly::{EvalContext, Requirement,
             TogglyClient};

pub async fn gate(client: &TogglyClient,
                  context: EvalContext)
  → toggly::Result<bool>
{
  client.evaluate_gate(
    &["new-dashboard", "api-v2"],
    Requirement::All, context, false,
  ).await
}
```

Any permits at least one true flag; All requires every flag. Negation applies to each result before combining. An empty list is false.

Order mapper

```
use std::collections::HashMap;
use toggly::{register_context, map_entity,
  EvalContext, TogglyEntityContext};

pub struct Order {
  pub id: String,
  pub vip: bool,
}

pub fn register_order() {
  register_context("Order", |value| {
    let order = value.downcast_ref::<Order>()
      .expect("Order mapper input");
    TogglyEntityContext {
      kind: "Order".into(),
      key: order.id.clone(),
      attributes: HashMap::from([
        ("Id".into(),
          serde_json::json!(order.id)),
        ("Vip".into(),
          serde_json::json!(order.vip)),
      ]),
    }
  }, None);
}

pub fn for_order(user_id: &str, order: &Order)
  -> EvalContext
{
  EvalContext::builder().identity(user_id)
    .entity(map_entity("Order", order)
      .expect("Order mapper registered"))
    .build()
}
```

Call register_order once before creating the client. None means no catalog schema upload. Pass for_order(...) to is_enabled("ExpressCheckout", ...); configure Order/Vip in Toggly. The guide shows the complete schema and optional Total.

Feature-key enum

```
use toggly::{EvalContext, TogglyClient};
use toggly_macros::FeatureFlags;

#[derive(FeatureFlags)]
pub enum Flags {
  #[toggly(key = "new-dashboard")]
  NewDashboard,
}

pub async fn check(client: &TogglyClient,
                   context: EvalContext)
  -> toggly::Result<bool>
{
  Flags::NewDashboard
    .is_enabled(client, context).await
}
```

The derive attribute is toggly. default_value() is metadata; it does not override an undefined-flag evaluation.

Function gate

```
use toggly::{EvalContext, TogglyClient};
use toggly_macros::feature_flag;

#[feature_flag(
  feature = "new-dashboard", client = "client",
  context = "context", fallback = "false"
)]
pub async fn show_new(client: &TogglyClient,
                      context: EvalContext) -> bool
{
  true
}
```

The attribute converts evaluation errors to false, then returns the fallback when gated. Use the direct client Result when errors must propagate.

Actix: shared Data and native middleware

```
use actix_web::{web, App, HttpResponse,
                HttpServer};
use toggly::TogglyClient;
use toggly_actix::{Feature, TogglyMiddleware};

async fn index(f: Feature) -> HttpResponse {
  if f.is_enabled("new-dashboard").await {
    HttpResponse::Ok().body("New dashboard")
  } else {
    HttpResponse::Ok().body("Classic dashboard")
  }
}

pub async fn serve(client: TogglyClient)
  -> std::io::Result<()>
{
  let owner = web::Data::new(client);
  let workers = owner.clone();
  let bound = HttpServer::new(move || {
    App::new().app_data(workers.clone())
      .service(web::resource("/")
        .wrap(TogglyMiddleware::with_feature(
          "api-v2"))
        .route(web::get().to(index)))
  }).bind(("127.0.0.1", 8080));
  let result = match bound {
    Ok(server) => server.run().await,
    Err(error) => Err(error),
  };
  owner.close().await;
  result
}
```

Use actix-web >=4.4,<4.16 with toggly-actix0.6; current tested host is Actix-web 4.15. Clone Data, not TogglyClient. with_feature is the native factory; negate() reverses the check. FeatureGuard tests definition existence, not enabled state.

Axum: native Extension and layer

```
use axum::{Extension, Router, routing::get};
use std::sync::Arc;
use toggly::TogglyClient;
use toggly_axum::{Feature, TogglyLayer};

async fn index(f: Feature) -> &'static str {
  if f.is_enabled("new-dashboard").await {
    "New dashboard"
  } else { "Classic dashboard" }
}

pub fn routes(client: Arc<TogglyClient>) -> Router {
  Router::new()
    .route("/", get(index))
    .route("/api/v2", get(|| async { "API v2" })
      .layer(TogglyLayer::require("api-v2")))
    .layer(Extension(client))
}
```

Use axum0.7 with toggly-axum0.6. Retain an Arc owner, drain the server on shutdown, then close().await. Typed State<TogglyState> is separate from the Extension required by native helpers.

Axum 0.8

```
# Cargo.toml
axum = "0.8"
toggly-axum08 = "0.1"

// imports
use toggly_axum08::{Feature, TogglyLayer, TogglyState
```

Use toggly-axum08 only with Axum 0.8. Keep toggly-axum for Axum 0.7; the packages stay separate because their Axum dependency types are incompatible.

Context at native boundaries

Feature helpers map x-user-id, then x-identity. Actix middleware reads only its configured identity_header. Axum layers check their configured identity header, then fall back to x-user-id, then x-identity when the preceding header is absent. These inputs are not authentication. They do not automatically map claims, groups, request segments or entities. Use TogglyData / TogglyExtractor or Feature.client() for complete explicit contexts; see each guide for runnable setup and shutdown.

Rocket: native guard and shutdown

```
use rocket::{fairing::AdHoc, get, routes,
             Build, Rocket, http::Status};
use toggly::{TogglyClient, TogglyConfig};
use toggly_rocket::{Feature, TogglyFairing};

#[get("/api/v2")]
async fn api(f: Feature<'_>)
  -> Result<&'static str, Status>
{
  if f.is_enabled("api-v2").await {
    Ok("API v2")
  } else { Err(Status::NotFound) }
}

pub fn application(config: TogglyConfig)
  -> Rocket<Build>
{
  rocket::build()
    .attach(TogglyFairing::from_config(config))
    .attach(AdHoc::on_shutdown("Close Toggly",
      |rocket| Box::pin(async move {
        if let Some(client) =
          rocket.state::<TogglyClient>()
        {
          client.close().await;
        }
      })))
    .mount("/", routes![api])
}
```

Use rocket0.5 with toggly-rocket0.6. The fairing initializes only; add the shutdown hook. The application returns 404 on false. FeatureEnabled/FeatureDisabled are not native request guards.

Usage and business metrics

```
use toggly::{MetricsFeatureOptions, TogglyClient};

pub fn conversion(client: &TogglyClient, id: &str) {
  client.record_view(
    "enhanced-submit", Some(id), "enabled");
  client.record_usage(
    "enhanced-submit", Some(id), "enabled");
  let opts = MetricsFeatureOptions::new(
    "enhanced-submit", Some("enabled"));
  client.measure("order-value", 120.0,
                 Some(&opts));
  client.increment_counter("orders", 1.0,
                           Some(&opts));
  client.observe("duration-ms", 350.0,
                 Some(&opts));
}
```

Enable toggly's telemetry Cargo feature and enable_usage_tracking(true)/enable_metrics(true) at startup. Record views and uses at their actual UI/action events. The function groups signatures only. Flush with flush_telemetry().await; close flushes best-effort. Labels are caller-supplied, not variant assignment.

Offline testing and API boundaries

Use a localhost definitions_url and real FeatureDefinition JSON, disable background refresh/live updates/schema upload/telemetry, and exercise native framework requests. A placeholder key with the remote default URL is not offline. Rust has no external custom evaluator trait, variant allocator or evaluated-snapshot import/export API. Passing a new context does not request fresh definitions or invalidate a retained decision.

Reference guides

Quick start · Order mapping · Actix · Axum · Rocket · Offline tests · Samples catalog