Advanced Rust - Lab: Async Rate Limiter Library

Lukáš Hozda

2025

Introduction

In this lab, you will build a practical async Rust library for rate limiting. Rate limiters are essential components in networked applications, allowing you to control the pace at which operations occur to prevent overloading APIs or services.

Setup

1. Create a new library project: `cargo new --lib async_rate_limiter`
2. Add required dependencies to your Cargo.toml:

```toml
[dependencies]
tokio = { version = "1", features = ["full"] }
thiserror = "1.0"
# Add any other dependencies you need
```

Exercise: Implementing an Async Rate Limiter (90 minutes)

Objective

Implement a thread-safe, async rate limiter library using Tokio that can control the rate of operations in concurrent environments. Your library should support both individual rate limiters and a registry for managing multiple limiters.

Instructions

Create a library that provides:

1. A token bucket rate limiting algorithm
2. Async-aware API that integrates well with Tokio applications
3. A registry for managing multiple rate limiters

Requirements

Core Rate Limiter

- Implement a ‘RateLimiter‘ struct with the following API:

```rust
pub struct RateLimiter {
    // Implementation details hidden from users
}

#[derive(Debug, thiserror::Error)]
```

```rust
pub enum Error {
    #[error("Rate limit exceeded and timed out waiting")]
    Timeout,
    // Add any other error types as needed
}

impl RateLimiter {
    /// Creates a new rate limiter
    ///
    /// # Parameters
    /// - `permits_per_second`: How many operations to allow per second
    /// - `max_burst`: Maximum number of operations allowed in a burst
    pub fn new(permits_per_second: f64, max_burst: u32) -> Self;

    /// Acquires a permit, waiting if necessary
    pub async fn acquire(&self) -> Result<(), Error>;

    /// Tries to acquire a permit without waiting
    pub async fn try_acquire(&self) -> bool;

    /// Acquires a permit, waiting up to the specified timeout
    pub async fn acquire_timeout(&self, timeout: Duration) -> Result<(), Error>;
}
```

Rate Limiter Registry

- Implement a ‘RateLimiterRegistry‘ for managing multiple named rate limiters:

```rust
pub struct RateLimiterRegistry {
    // Implementation details hidden from users
}

impl RateLimiterRegistry {
    /// Creates a new empty registry
    pub fn new() -> Self;

    /// Registers a rate limiter with the given name
    pub fn register<S: Into<String>>(&self, name: S, limiter: RateLimiter) -> Arc<RateLimiter>;

    /// Gets a rate limiter by name
    pub fn get<S: AsRef<str>>(&self, name: S) -> Option<Arc<RateLimiter>>;

    /// Gets a rate limiter by name, or creates a new one using the provided function
    pub fn get_or_create<S, F>(&self, name: S, create_fn: F) -> Arc<RateLimiter>
    where
        S: Into<String>,
        F: FnOnce() -> RateLimiter;

    /// Removes a rate limiter by name
    pub fn remove<S: AsRef<str>>(&self, name: S) -> Option<Arc<RateLimiter>>;
}
```

Testing & Documentation

- Write tests for your library

- Document all public APIs with rustdoc comments
- Include examples of how to use the library
- Ensure thread safety for concurrent applications

Example Usage

Single Rate Limiter Example

```rust
use async_rate_limiter::RateLimiter;
use std::time::Duration;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Create a rate limiter that allows 5 operations per second with burst of 1
    let limiter = RateLimiter::new(5.0, 1);

    for i in 0..10 {
        // Will automatically wait if rate limit exceeded
        limiter.acquire().await?;
        println!("Operation {}", i);

        // Simulate work
        tokio::time::sleep(Duration::from_millis(50)).await;
    }

    Ok(())
}
```

Registry Example for Web API

```rust
use async_rate_limiter::{RateLimiter, RateLimiterRegistry};
use std::sync::Arc;

#[tokio::main]
async fn main() {
    // Create registry
    let registry = Arc::new(RateLimiterRegistry::new());

    // Register different endpoint limiters
    registry.register("login", RateLimiter::new(5.0, 0));      // 5 req/sec
    registry.register("public_api", RateLimiter::new(100.0, 20)); // 100 req/sec

    // In a web handler (pseudocode)
    async fn handle_login(registry: Arc<RateLimiterRegistry>, client_ip: String) {
        // Get endpoint limiter
        let endpoint_limiter = registry.get("login").unwrap();

        // Get per-IP limiter (create if doesn't exist)
        let ip_limiter = registry.get_or_create(client_ip, || {
            RateLimiter::new(2.0, 1)  // Only 2 login attempts per second per IP
        });

        // Apply both limits
        if endpoint_limiter.try_acquire().await && ip_limiter.try_acquire().await {
            // Process login request
```

```
        } else {
            // Return rate limit exceeded error
        }
    }
}
```

Getting Started

Here's a skeleton to begin with:

```
// src/lib.rs
use std::sync::Mutex;
use std::time::{Duration, Instant};
use tokio::time::sleep;

#[derive(Debug, thiserror::Error)]
pub enum Error {
    #[error("Rate limit exceeded and timed out waiting")]
    Timeout,
}

pub struct RateLimiter {
    // TODO: Implement fields
}

struct RateLimiterState {
    // TODO: Implement state fields
}

impl RateLimiter {
    pub fn new(permits_per_second: f64, max_burst: u32) -> Self {
        todo!()
    }

    pub async fn try_acquire(&self) -> bool {
        todo!()
    }

    pub async fn acquire(&self) -> Result<(), Error> {
        todo!()
    }

    pub async fn acquire_timeout(&self, timeout: Duration) -> Result<(), Error> {
        todo!()
    }
}

pub struct RateLimiterRegistry {
    // Implementation details hidden from users
}

impl RateLimiterRegistry {
    /// Creates a new empty registry
    pub fn new() -> Self {
        todo!()
```

```
    }

    /// Registers a rate limiter with the given name
    pub fn register<S: Into<String>>(&self, name: S, limiter: RateLimiter) -> Arc<RateLimiter> {
        todo!()
    }

    /// Gets a rate limiter by name
    pub fn get<S: AsRef<str>>(&self, name: S) -> Option<Arc<RateLimiter>> {
        todo!()
    }

    /// Gets a rate limiter by name, or creates a new one using the provided function
    pub fn get_or_create<S, F>(&self, name: S, create_fn: F) -> Arc<RateLimiter>
    where
        S: Into<String>,
        F: FnOnce() -> RateLimiter
    {
        todo!()
    }

    /// Removes a rate limiter by name
    pub fn remove<S: AsRef<str>>(&self, name: S) -> Option<Arc<RateLimiter>> {
        todo!()
    }
}
```