


# Corso: Programmazione di sistema

## Quiz: API Programming - Rust 17 giugno 2025



PORTALE  
ESAMI

	Umberto Fontanazza <b>s323916</b>
<b>Iniziato</b>	17 giugno 2025, 16:37
<b>Stato</b>	Completato
<b>Terminato</b>	17 giugno 2025, 17:37
<b>Tempo impiegato</b>	1 ora
<b>Valutazione</b>	<b>5,0</b> su un massimo di 6,0 ( <b>83%</b> )
<b>Riepilogo del tentativo</b>	<div><div>1</div></div>

### Domanda 1

Completo

Punteggio  
ottenuto 5,0  
su 6,0

### TokenManager



Un applicativo software multithread fa accesso ai servizi di un server remoto, attraverso richieste di tipo HTTP.

Tali richieste devono includere un token di sicurezza che identifica l'applicativo stesso e ne autorizza l'accesso.

Per motivi di sicurezza, il token ha una validità limitata nel tempo (qualche minuto) e deve essere rinnovato alla sua scadenza.

Il token viene ottenuto attraverso una funzione (fornita esternamente e conforme al tipo `TokenAcquirer`) che restituisce alternativamente un token e la sua data di scadenza o un messaggio di errore se non è possibile fornirlo.

Poiché la emissione richiede un tempo apprezzabile (da alcune centinaia di millisecondi ad alcuni secondi), si vuole centralizzare la gestione del token, per evitare che più thread ne facciano richiesta in contemporanea. A tale scopo deve essere implementata la `struct TokenManager` che si occupa di gestire il rilascio, il rinnovo e la messa a disposizione del token a chi ne abbia bisogno, secondo la logica di seguito indicata.

La `struct TokenManager` offre i seguenti metodi:

```
type TokenAcquirer = dyn Fn() -> Result<(String, Instant), String> + Sync

pub fn new(acquire_token: Box<TokenAcquirer> ) -> Self
pub fn get_token(&self) -> Result<String, String>s
pub fn try_get_token(&self) -> Option<string>
```

Al proprio interno, la `struct TokenManager` mantiene 3 possibili stati:

1. `Empty` - indica che non è ancora stato richiesto alcun token
2. `Pending` - indica che è in corso una richiesta di acquisizione del token
3. `Valid` - indica che è disponibile un token in corso di validità

Il metodo `new(...)` riceve il puntatore alla funzione in grado di acquisire il token. Essa opera in modalità pigra e si limita a creare un'istanza della struttura con le necessarie informazioni per gestire il suo successivo comportamento.

Il metodo `get_token(...)` implementa il seguente comportamento:

- Se lo stato è `Empty`, passa allo stato `Pending` e invoca la funzione per acquisire il token; se questa ritorna un risultato valido, memorizza il token e la sua scadenza, porta lo stato a `Valid` e restituisce copia del token stesso; se, invece, questa restituisce un errore, pone lo stato a `Empty` e restituisce l'errore ricevuto.
- Se lo stato è `Pending`, attende senza consumare cicli di CPU che questo passi ad un altro valore, dopodiché si comporta di conseguenza.
- Se lo stato è `Valid` e il token non risulta ancora scaduto, ne restituisce una copia; altrimenti pone lo stato ad `Pending` e inizia una richiesta di acquisizione, come indicato sopra.

Il metodo `try_get_token(...)` implementa il seguente comportamento:

- Se lo stato è `Valid` e il token non è scaduto, restituisce una copia del token opportunamente incapsulata in un oggetto di tipo `Option`.

- In tutti gli altri casi restituisce **None**.

Si implementi tale struttura nel linguaggio Rust.

A supporto della validazione del codice realizzato si considerino i seguenti test (due dei quali sono forniti con la relativa implementazione, i restanti sono solo indicati e devono essere opportunamente completati):

```
[test]
fn a_new_manager_contains_no_token() {
    let a: Box<tokenacquirer> = Box::new(|| Err("failure".to_string()));
    let manager = TokenManager::new(a);
    assert!(manager.try_get_token().is_none());
}

[test]
fn a_failing_acquirer_always_returns_an_error() {
    let a: Box<tokenacquirer> = Box::new(|| Err("failure".to_string()));
    let manager = TokenManager::new(a);
    assert_eq!(manager.get_token(), Err("failure".to_string()));
    assert_eq!(manager.get_token(), Err("failure".to_string()));
}

[test]
fn a_successful_acquirer_always_returns_success() {
    //...to be implemented
}

[test]
fn a_slow_acquirer_causes_other_threads_to_wait() {
    //...to be implemented
}
```

*La risposta a questa domanda è stata inserita tramite un'istanza Crownlabs ed è visibile sul Portale della Didattica sulla sezione **Consegna Elaborati** di questo insegnamento.*

Commento:

La funzione di acquisizione del token è immutabile e non deve stare all'interno del Mutex: infatti, mentre si acquisisce il token, occorre rilasciare il mutex, altrimenti lo stato Pending non è osservabile.

La notifica va inviata quando lo stato diventa Empty o Valid a seguito dell'invocazione della funzione di acquisizione.

Piuttosto che costruire una funzione ricorsiva, di cui non puoi controllare la profondità, è opportuno trasformarla in una funzione basata su un ciclo, così da poter gestire l'evoluzione dello stato.

Una soluzione basata sul tuo codice 'è:

```

use std::time::{Duration, Instant};
use std::sync::{Mutex, Condvar, Arc};
use std::thread::sleep;

type TokenAcquirer = dyn Fn() -> Result<(String, Instant), String> + Sync+Send;

#[derive(Clone, PartialEq)]
enum State {
    Empty,
    Pending, // with this implementation Pending is basically useless
    Valid((String, Instant))
}

pub struct TokenManager {
    mutex: Mutex<State>,
    condvar: Condvar,
    acquire_token: Box<TokenAcquirer>,
}

impl TokenManager {
    pub fn new(acquire_token: Box<TokenAcquirer>) -> Self {
        Self {
            mutex: Mutex::new(State::Empty),
            condvar: Condvar::new(),
            acquire_token: acquire_token,
        }
    }

    pub fn get_token(&self) -> Result<String, String> {
        let mut lock = self.mutex.lock().unwrap();
        loop {
            match &*lock {
                State::Empty => {
                    *lock = State::Pending;
                    drop(lock);
                    let res = (self.acquire_token)();
                    lock = self.mutex.lock().unwrap();
                    match res {
                        Ok((token, expiration)) => {
                            *lock = State::Valid((token.clone(), expiration));
                            drop(lock);
                            self.condvar.notify_all();
                            return Ok(token)
                        },
                        Err(e) => {
                            *lock = State::Empty;
                            drop(lock);
                            self.condvar.notify_all();
                            return Err(e)
                        },
                    }
                },
                State::Pending => {
                    lock = self.condvar.wait_while(lock, |l| *l == State::Pending).unwrap();
                },
                State::Valid((token, expiration)) => {
                    if *expiration > Instant::now() {
                        return Ok(token.clone())
                    } else {
                        *lock = State::Empty
                    }
                }
            }
        }
    }

    pub fn try_get_token(&self) -> Option<String> {

```

```
        let lock = self.mutex.lock().unwrap();
        if let State::Valid((token, expiration)) = &*lock {
            if *expiration > Instant::now() {
                return Some(token.clone());
            }
        }
        None
    }
}
```