**Streamed Cache Assignment**

use std::collections::HashMap;

use std::sync::{Arc, Mutex};

use std::thread;

use std::time::Duration;

struct TemperatureApi {

// Simulating the actual API structure

// You can replace this with the actual API structure and implementation

// For simplicity, let's assume cities are identified by their names.

temperatures: HashMap<String, i32>,

subscribers: Vec<Box<dyn Fn(&HashMap<String, i32>) + Send>>,

}

impl TemperatureApi {

fn new() -> Self {

TemperatureApi {

temperatures: HashMap::new(),

subscribers: Vec::new(),

}

}

fn fetch\_all\_temperatures(&self) -> HashMap<String, i32> {

// Simulating the slow and expensive fetch call

self.temperatures.clone()

}

fn subscribe(&mut self, callback: Box<dyn Fn(&HashMap<String, i32>) + Send>) {

self.subscribers.push(callback);

}

fn start\_update\_thread(api: Arc<Mutex<Self>>) {

thread::spawn(move || {

loop {

// Simulating periodic updates

thread::sleep(Duration::from\_secs(5));

// In a real scenario, you would use the actual API to get incremental updates

let mut api = api.lock().unwrap();

api.temperatures.insert("City1".to\_string(), rand::random::<i32>());

api.temperatures.insert("City2".to\_string(), rand::random::<i32>());

// Notify subscribers about the updated temperatures

for subscriber in api.subscribers.iter() {

subscriber(&api.temperatures);

}

}

});

}

}

struct TemperatureCache {

api: Arc<Mutex<TemperatureApi>>,

}

impl TemperatureCache {

fn new(api: TemperatureApi) -> Self {

let api = Arc::new(Mutex::new(api));

// Start the background thread for updates

TemperatureApi::start\_update\_thread(api.clone());

TemperatureCache { api }

}

fn get\_temperature(&self, city: &str) -> i32 {

let api = self.api.lock().unwrap();

if let Some(&temperature) = api.temperatures.get(city) {

temperature

} else {

// City not found, you may want to handle this case accordingly

0

}

}

fn subscribe\_to\_updates(&self, callback: Box<dyn Fn(&HashMap<String, i32>) + Send>) {

let mut api = self.api.lock().unwrap();

api.subscribe(callback);

}

}

#[cfg(test)]

mod tests {

use super::\*;

#[test]

fn test\_temperature\_cache() {

let api = TemperatureApi::new();

let cache = TemperatureCache::new(api);

let temperature = cache.get\_temperature("City1");

assert\_eq!(temperature, 0); // Initially, the temperature is not available

let result = Arc::new(Mutex::new(None));

let result\_clone = result.clone();

cache.subscribe\_to\_updates(Box::new(move |temperatures| {

let mut result = result\_clone.lock().unwrap();

\*result = Some(temperatures.clone());

}));

// Sleep to allow the background thread to update temperatures

thread::sleep(Duration::from\_secs(10));

let result = result.lock().unwrap();

assert\_eq!(result.is\_some(), true);

let temperatures = result.as\_ref().unwrap();

assert\_eq!(temperatures.contains\_key("City1"), true);

}

}

fn main() {

// You can run the tests using `cargo test`

}