



Haidaraly Rémy 4SI4

Sommaire :

Sommaire :

Objectif 0 :

Objectif 1 :

Objectif 2 :

Objectif 3 :

Objectif 4 :

Objectif 5 :

Objectif 6 :

Objectif 7 :

Objectif 8 :

Objectif 9 :

Objectif 0 :

Faire une requête GET sur un serveur python.

```
python3 -m http.server 8000
```

Rust :

```
use request::Error;

async fn make_request() -> Result<(), Error> {
    let client = request::Client::new();
```

```

    let url = "http://localhost:8000";

    // Envoyez une requête GET à l'URL
    let response = client.get(url).send().await?;

    // Vérifiez si la réponse est réussie (code de statut 200)
    if response.status().is_success() {
        println!("Requête GET réussie !");
    } else {
        println!("La requête GET a échoué avec le code de statut : {}", response.status());
    }

    Ok(())
}

// Fonction principale
#[tokio::main]
async fn main() -> Result<(), Error> {
    make_request().await?;

    Ok(())
}

```

Cargo.toml :

```

[package]
name = "fxbalancer"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

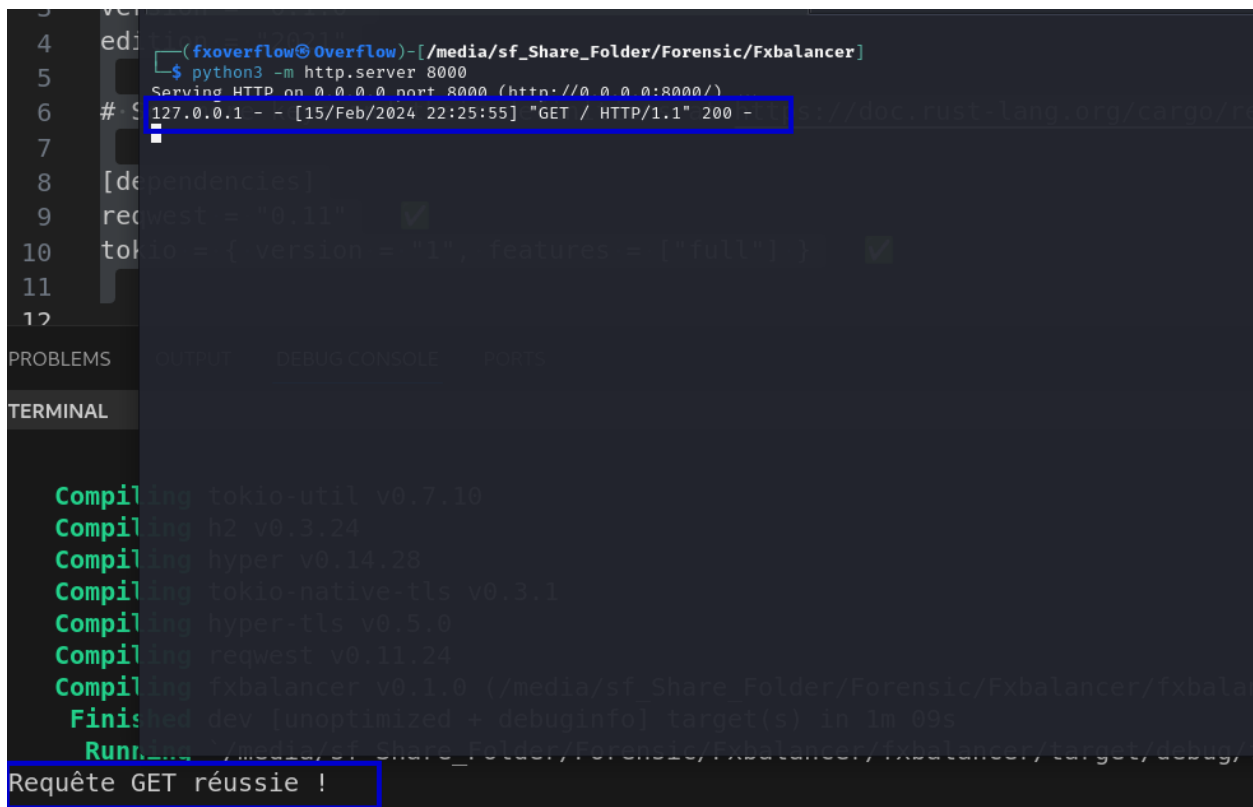
```

```
g.org/cargo/reference/manifest.html
```

```
[dependencies]
```

```
request = "0.11"
```

```
tokio = { version = "1", features = ["full"] }
```



```
(fxoverflow@Overflow)-[/media/sf_Share_Folder/Forensic/Fxbalancer]
$ python3 -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/)
# $ 127.0.0.1 - - [15/Feb/2024 22:25:55] "GET / HTTP/1.1" 200 -
[dependencies]
request = "0.11"
tokio = { version = "1", features = ["full"] }

PROBLEMS OUTPUT DEBUG CONSOLE PORTS
TERMINAL

Compiling tokio-util v0.7.10
Compiling h2 v0.3.24
Compiling hyper v0.14.28
Compiling tokio-native-tls v0.3.1
Compiling hyper-tls v0.5.0
Compiling request v0.11.24
Compiling fxbalancer v0.1.0 (/media/sf_Share_Folder/Forensic/Fxbalancer/fxbalancer)
Finished dev [unoptimized + debuginfo] target(s) in 1m 09s
Running /media/sf_Share_Folder/Forensic/Fxbalancer/fxbalancer/target/debug/fxbalancer
Requête GET réussie !
```

async / await : En Rust, `async` et `await` sont utilisés pour la programmation asynchrone. La programmation asynchrone permet d'effectuer plusieurs tâches en parallèle sans bloquer l'exécution du programme. Dans notre cas, la fonction `make_request` est déclarée comme étant asynchrone avec `async fn make_request()`. Cela signifie que cette fonction peut effectuer des opérations de manière asynchrone, ce qui est nécessaire pour envoyer une requête HTTP de manière non bloquante.

await : `await` est utilisé pour attendre que le résultat d'une opération asynchrone soit disponible. Dans notre cas, `client.get(url).send().await?` envoie une requête

GET de manière asynchrone à l'URL spécifiée et attend que la réponse soit disponible. L'utilisation de `await` permet de suspendre temporairement l'exécution de la fonction jusqu'à ce que la réponse soit reçue.

Objectif 1 :

Créer un reverse proxy qui redirige le client sur le serveur python !

On va utiliser la bibliothèque **actix-web** pour créer un serveur web qui peut recevoir des demandes de clients.

Exemple de simple code qui crée un serveur web en rust et print **Hello, Actix Web** !

```
use request::Error;
use actix_web::{App, HttpServer, HttpResponse, web};

async fn index() -> HttpResponse {
    HttpResponse::Ok().body("Hello, Actix Web!")
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(index))
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}
```

Rediriger un client sur un serveur python :

```

use actix_web::{App, HttpServer, HttpResponse, web};

async fn redirect() -> HttpResponse {
    let redirect_url = "http://127.0.0.1:8000";
    HttpResponse::TemporaryRedirect()
        .append_header(("Location", redirect_url))
        .finish()
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(redirect))
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}

```

Je dois utiliser la commande suivante :

```

curl -L http://127.0.0.1:8080

# -L pour dire oui pour la redirection !

```

Maintenant on va faire pareille pour mais un vecteur de serveur python ou le reverse proxy prendra aléatoirement un serveur de ce vecteur.

Cargot.toml

```
rand = "0.8"
request = "0.11"
tokio = { version = "1", features = ["full"] }
```

Code :

```
use actix_web::{App, HttpServer, HttpResponse, web};
use rand::seq::SliceRandom;

async fn redirect() -> HttpResponse {
    // Liste de serveurs Python disponibles
    let servers = vec![
        "http://127.0.0.1:8000",
        "http://127.0.0.1:8001",
    ];

    // Choix aléatoire d'un serveur dans la liste
    let chosen_server = servers.choose(&mut rand::thread_rng
    ()).unwrap();

    // Redirection vers le serveur choisi
    HttpResponse::TemporaryRedirect()
        .append_header(("Location", chosen_server.to_string
    ()))
        .finish()
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(redirect))
    })
    .bind("127.0.0.1:8080")?
```

```
.run()  
.await  
}
```

```
(fxoverflow@Overflow)-[~/Desktop/server1]  
$ curl -L http://127.0.0.1:8080  
server 1  
(fxoverflow@Overflow)-[~/Desktop/server1]  
$ curl -L http://127.0.0.1:8080  
server 2  
(fxoverflow@Overflow)-[~/Desktop/server1]  
$ curl -L http://127.0.0.1:8080  
server 2  
(fxoverflow@Overflow)-[~/Desktop/server1]  
$ curl -L http://127.0.0.1:8080  
server 2  
(fxoverflow@Overflow)-[~/Desktop/server1]  
$ curl -L http://127.0.0.1:8080  
server 1  
(fxoverflow@Overflow)-[~/Desktop/server1]  
$ curl -L http://127.0.0.1:8080  
server 2  
(fxoverflow@Overflow)-[~/Desktop/server1]  
$ curl -L http://127.0.0.1:8080  
server 1  
(fxoverflow@Overflow)-[~/Desktop/server1]  
$
```

Cependant si on coupe un serveur par exemple le 2 :

```
(fxoverflow@Overflow)-[~/Desktop/server1]  
$ curl -L http://127.0.0.1:8080  
server 1  
(fxoverflow@Overflow)-[~/Desktop/server1]  
$ curl -L http://127.0.0.1:8080  
curl: (7) Failed to connect to 127.0.0.1 port 8001 after 0 ms: Couldn't connect to server  
(fxoverflow@Overflow)-[~/Desktop/server1]  
$
```

Je vais avoir une erreur ... il va donc falloir faire en sorte que le serveur python puisse informé notre reverse proxy s'il est **alive** ou non et en fonction notre reverse proxy nous enverrons sur le bon serveur alive !

Objectif 2 :

Gérer l'erreur rencontrer dans l'objectif 2 !

Cargo.toml

```
actix-web = "4.0.0"
rand = "0.8.4"
awc = "3.4.0"
```

main.rs

```
use actix_web::{web, App, HttpResponse, HttpServer, http::StatusCode};
use awc::Client;
use rand::seq::SliceRandom;

async fn redirect() -> HttpResponse {
    let servers = vec![
        "http://127.0.0.1:8000",
        "http://127.0.0.1:8001",
    ];

    let client = Client::default();

    // Choix aléatoire d'un serveur dans la liste
    let chosen_server = *servers.choose(&mut rand::thread_rng()).unwrap();

    // Vérification si le serveur est vivant avec une requête
```


HTTP

```
let response = client.get(chosen_server)
    .send()
    .await;

match response {
    Ok(resp) if resp.status().is_success() => {
        // Si le serveur est vivant, rediriger vers le se
rveur choisi
        HttpResponse::TemporaryRedirect()
            .append_header(("Location", chosen_server))
            .finish()
    },
    _ => {
        // Si le serveur est mort ou la requête a échoué,
renvoyer une erreur 502 Bad Gateway
        HttpResponse::new(StatusCode::BAD_GATEWAY)
    },
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new().route("/", web::get().to(redirect))
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}
```

Ici avant que le proxy redirige on teste le serveur python pour voir s'il est vivant si oui alors on redirige mais si ça fonctionne pas alors on envoie une 502.

```
(fxoverflow@Overflow)-[~/Desktop/server1]
$ python3 -m http.server 8001
Serving HTTP on 0.0.0.0 port 8001 (http://0.0.0.0:8001/) ...
127.0.0.1 - - [20/Feb/2024 22:43:45] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:46] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:47] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:49] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:49] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:50] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:50] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:51] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:58] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:44:22] "GET / HTTP/1.1" 200 -

(fxoverflow@Overflow)-[~/Desktop/server1]
$ curl -v http://127.0.0.1:8080
Trying 127.0.0.1:8080...
Connected to 127.0.0.1 (127.0.0.1) port 8080
GET / HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: curl/8.5.0
Accept: */*

HTTP/1.1 502 Bad Gateway
content-length: 0
date: Tue, 20 Feb 2024 21:43:56 GMT

Connection #0 to host 127.0.0.1 left intact

(fxoverflow@Overflow)-[~/Desktop/server1]
$ curl -v https://127.0.0.1:8080
Trying 127.0.0.1:8080...
Connected to 127.0.0.1 (127.0.0.1) port 8080
GET / HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: curl/8.5.0
Accept: */*

HTTP/1.1 307 Temporary Redirect
content-length: 0
location: http://127.0.0.1:8001
date: Tue, 20 Feb 2024 21:43:58 GMT

Connection #0 to host 127.0.0.1 left intact

(fxoverflow@Overflow)-[~/Desktop/server2]
$ python3 -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
127.0.0.1 - - [20/Feb/2024 22:43:41] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:46] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:48] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:51] "GET / HTTP/1.1" 200 -
Keyboard interrupt received, exiting.

(fxoverflow@Overflow)-[~/Desktop/server2]
$ python3 -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
127.0.0.1 - - [20/Feb/2024 22:44:24] "GET / HTTP/1.1" 200 -
```

Comme on peut le voir sur cette image, ça fonctionne si je down un serveur python j'aurai une **erreur 502**

Si je réactive le serveur python :

```
(fxoverflow@Overflow)-[~/Desktop/server1]
$ python3 -m http.server 8001
Serving HTTP on 0.0.0.0 port 8001 (http://0.0.0.0:8001/) ...
127.0.0.1 - - [20/Feb/2024 22:43:45] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:46] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:47] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:49] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:49] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:50] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:50] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:51] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:43:58] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Feb/2024 22:44:22] "GET / HTTP/1.1" 200 -

(fxoverflow@Overflow)-[~/Desktop/server1]
$ curl -v http://127.0.0.1:8080
Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080
> GET / HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/8.5.0
> Accept: */*
>
< HTTP/1.1 307 Temporary Redirect
< content-length: 0
< location: http://127.0.0.1:8001
< date: Tue, 20 Feb 2024 21:44:22 GMT
<
* Connection #0 to host 127.0.0.1 left intact

(fxoverflow@Overflow)-[~/Desktop/server1]
$ curl -v https://127.0.0.1:8080
Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080
> GET / HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/8.5.0
> Accept: */*
>
< HTTP/1.1 307 Temporary Redirect
< content-length: 0
< location: http://127.0.0.1:8000
< date: Tue, 20 Feb 2024 21:44:24 GMT
<
* Connection #0 to host 127.0.0.1 left intact

(fxoverflow@Overflow)-[~/Desktop/server1]
$ curl -v http://127.0.0.1:8080
```

Comme on peut le voir, il va de nouveau me redirigé !



Maintenant il faut trouver un moyen pour ne pas avoir une erreur 502 mais plutôt de rediriger tout le trafic sur le seul serveur vivant, et si tous les serveurs sont down alors j'envoie une 502 !

Objectif 3 :

Répondre à la problématique précédentes !

```
use actix_web::{web, App, HttpResponse, HttpServer, http::Sta
tusCode};
use awc::Client;
use rand::seq::SliceRandom;
use std::sync::Mutex;
use lazy_static::lazy_static;

lazy_static! {
    // Utilisation d'un Mutex pour protéger l'accès concurren
    t aux serveurs en amont
    static ref UPSTREAM_SERVERS: Mutex<Vec<(String, bool)>> =
    Mutex::new(vec![
        ("http://127.0.0.1:8000".to_string(), true),
        ("http://127.0.0.1:8001".to_string(), true),
    ]);
}

async fn try_upstream(server: &str, client: &Client) -> Resul
t<HttpResponse, ()> {
    let response = client.get(server).send().await;
```

```

        match response {
            Ok(resp) if resp.status().is_success() => Ok(HttpResponse::Ok().body(format!("Proxied to {}", server))),
            _ => Err(()),
        }
    }
}

async fn redirect() -> HttpResponse {
    let client = Client::default();
    let mut servers = UPSTREAM_SERVERS.lock().unwrap();

    // Sélection aléatoire d'un serveur
    let mut rng = rand::thread_rng();
    let servers_order: Vec<usize> = (0..servers.len()).collect();
    let mut servers_order_shuffled = servers_order.clone();
    servers_order_shuffled.shuffle(&mut rng);

    for index in servers_order_shuffled {
        let (server, alive) = &servers[index];
        if *alive {
            match try_upstream(server, &client).await {
                Ok(response) => return response,
                Err(_) => {
                    // Marquer le serveur comme mort
                    servers[index].1 = false;
                },
            }
        }
    }

    // Si tous les serveurs sont morts
    if servers.iter().all(|(_, alive)| !*alive) {
        HttpResponse::new(StatusCode::BAD_GATEWAY)
    } else {
        HttpResponse::new(StatusCode::SERVICE_UNAVAILABLE)
    }
}

```

```

    }
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new().route("/", web::get().to(redirect))
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}

```

Ici le code il détecte bien un serveur down il redirige le trafic au seul serveur vivant. Si je coupe tous les serveurs alors j'ai une erreur 502.

Problème :



Si je rallume le serveur down en up, le code ne le prends pas en compte....

Objectif 4 :

Répondre à la problématique, en faisant un check toutes les 10 secondes pour prendre en compte si un serveur est up ou down pour résoudre le problème, crée une fonction **check_upstream_servers()**

```

actix-web = "4.0.0"
rand = "0.8.4"
awc = "3.4.0"
lazy_static = "1.4.0"
futures = "0.3"

```

```
tokio = { version = "1.0", features = ["full"] }
actix-rt = "2.5"
```

rust.rs

```
use actix_web::{web, App, HttpResponse, HttpServer, http::Sta
tusCode};
use awc::Client;
use rand::seq::SliceRandom;
use std::sync::Mutex;
use lazy_static::lazy_static;
use std::time::Duration;
use actix_rt::time::interval;
```

```
lazy_static! {
    static ref UPSTREAM_SERVERS: Mutex<Vec<(String, bool)>> =
Mutex::new(vec![
        ("http://127.0.0.1:8000".to_string(), true),
        ("http://127.0.0.1:8001".to_string(), true),
    ]);
}
```

```
async fn try_upstream(server: &str, client: &Client) -> Resul
t<HttpResponse, ()> {
    let response = client.get(server).send().await;
    match response {
        Ok(resp) if resp.status().is_success() => Ok(HttpResp
onse::Ok().body(format!("Proxied to {}", server))),
        _ => Err(()),
    }
}
```

```
async fn redirect() -> HttpResponse {
```

```

let client = Client::default();
let mut servers = UPSTREAM_SERVERS.lock().unwrap();

let mut rng = rand::thread_rng();
let servers_order: Vec<usize> = (0..servers.len()).collect();
let mut servers_order_shuffled = servers_order.clone();
servers_order_shuffled.shuffle(&mut rng);

for index in servers_order_shuffled {
    let (server, alive) = &servers[index];
    if *alive {
        match try_upstream(server, &client).await {
            Ok(response) => return response,
            Err(_) => servers[index].1 = false,
        }
    }
}

if servers.iter().all(|(_, alive)| !*alive) {
    HttpResponse::new(StatusCode::BAD_GATEWAY)
} else {
    HttpResponse::new(StatusCode::SERVICE_UNAVAILABLE)
}
}

// Fonction pour vérifier l'état des serveurs en amont
async fn check_upstream_servers() {
    let client = Client::default();
    let mut servers = UPSTREAM_SERVERS.lock().unwrap();
    for (server, alive) in servers.iter_mut() {
        *alive = client.get(server.as_str()).send().await.is_ok();
    }
}

```

```
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    // Créez un intervalle pour vérifier l'état des serveurs
    toutes les 10 secondes
    let mut interval = interval(Duration::from_secs(10));
    actix_rt::spawn(async move {
        loop {
            interval.tick().await;
            check_upstream_servers().await;
        }
    });

    HttpServer::new(|| {
        App::new().route("/", web::get().to(redirect))
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}
```

Tout fonctionne !

Objectif 5 :

Amélioration mineures sur le code

Afficher quand les serveurs sont **up** ou **down** (de préférence en couleur)

Codes de couleur ANSI :

Vert pour les messages : \x1b[32m

Rouge pour les messages : \x1b[31m

Réinitialiser la couleur : \x1b[0m


```

use actix_web::{web, App, HttpResponse, HttpServer, http::Sta
tusCode};
use awc::Client;
use rand::seq::SliceRandom;
use std::sync::Mutex;
use lazy_static::lazy_static;
use std::time::Duration;
use actix_rt::time::interval;

lazy_static! {
    static ref UPSTREAM_SERVERS: Mutex<Vec<(String, bool)>> =
Mutex::new(vec![
        ("http://127.0.0.1:8000".to_string(), true),
        ("http://127.0.0.1:8001".to_string(), true),
    ]);
}

async fn try_upstream(server: &str, client: &Client) -> Resul
t<HttpResponse, ()> {
    let response = client.get(server).send().await;
    match response {
        Ok(resp) if resp.status().is_success() => Ok(HttpResp
onse::Ok().body(format!("Proxied to {}", server))),
        _ => Err(()),
    }
}

async fn redirect() -> HttpResponse {
    let client = Client::default();
    let mut servers = UPSTREAM_SERVERS.lock().unwrap();

    let mut rng = rand::thread_rng();
    let servers_order: Vec<usize> = (0..servers.len()).collec
t();

```

```

let mut servers_order_shuffled = servers_order.clone();
servers_order_shuffled.shuffle(&mut rng);

for index in servers_order_shuffled {
    let (server, alive) = &servers[index];
    if *alive {
        match try_upstream(server, &client).await {
            Ok(response) => return response,
            Err(_) => servers[index].1 = false,
        }
    }
}

HttpResponse::new(StatusCode::SERVICE_UNAVAILABLE)
}

async fn check_upstream_servers() {
    let client = Client::default();
    let mut servers = UPSTREAM_SERVERS.lock().unwrap();
    for (server, alive) in servers.iter_mut() {
        let prev_alive = *alive;
        *alive = client.get(server.as_str()).send().await.is_
ok();

        if *alive != prev_alive {
            print_server_status(server, *alive);
        }
    }

    print_servers_summary(&servers);
}

fn print_server_status(server: &String, is_up: bool) {
    let (color_code, status) = if is_up { ("\x1b[32m", "UP")
} else { ("\x1b[31m", "DOWN") };
    println!("{}", "Server {} is now {}", color_code, server, s

```

```

tatus, "\x1b[0m");
}

fn print_servers_summary(servers: &Vec<(String, bool)>) {
    let up_servers: Vec<String> = servers.iter().filter(|(_,
alive)| *alive).map(|(server, _)| server.clone()).collect();
    let down_servers: Vec<String> = servers.iter().filter(|
(_, alive)| !*alive).map(|(server, _)| server.clone()).collec
t();

    if !up_servers.is_empty() {
        println!("\x1b[32mListe des serveurs up {:?}\x1b[0m",
up_servers);
    }
    if !down_servers.is_empty() {
        println!("\x1b[31mListe des serveurs down {:?}\x1b[0
m", down_servers);
    }
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    let mut interval = interval(Duration::from_secs(10));
    actix_rt::spawn(async move {
        loop {
            interval.tick().await;
            check_upstream_servers().await;
        }
    });

    HttpServer::new(|| {
        App::new().route("/", web::get().to(redirect))
    })
    .bind("127.0.0.1:8080")?
    .run()

```

```
    .await
}
```

Résultat :

```
(f XavierFlow@OverFlow) ~/Desktop/server1
$ python3 -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
127.0.0.1 ~ [22/Feb/2024 23:26:12] "GET / HTTP/1.1" 200 -
127.0.0.1 ~ [22/Feb/2024 23:26:22] "GET / HTTP/1.1" 200 -
127.0.0.1 ~ [22/Feb/2024 23:26:32] "GET / HTTP/1.1" 200 -

(f XavierFlow@OverFlow) ~/Desktop/server2
$ python3 -m http.server 8001
Serving HTTP on 0.0.0.0 port 8001 (http://0.0.0.0:8001/) ...
127.0.0.1 ~ [22/Feb/2024 23:26:22] "GET / HTTP/1.1" 200 -
127.0.0.1 ~ [22/Feb/2024 23:26:32] "GET / HTTP/1.1" 200 -

Blocking waiting for file lock on build directory
Compiling balancer_0.c.o (/media/sf_Share_Folder/balancer)
warning: hard linking files in the incremental compilation cache failed; copying files instead; consider moving the cache dir
tem which supports hard linking in session dir: /media/sf_Share_Folder/balancer/target/debug/incremental/balancer-1fc64v47xe6w
a4-working`
warning: error deleting incremental compilation session directory "/media/sf_Share_Folder/balancer/target/debug/
r-1fc64v47xe6w6/s-gtns": I/O error: lock: text file busy (os error 26)
warning: error deleting lock file for incremental compilation session directory "/media/sf_Share_Folder/balancer/target/debug/
r-1fc64v47xe6w6/s-gtns": I/O error: lock: text file busy (os error 26)
warning: `balancer` (bin "balancer") generated 3 warnings
Finished dev [unoptimized + debuginfo] target(s) in 1m 02s
Running `target/debug/balancer`
Server http://127.0.0.1:8000 is now UP
Server http://127.0.0.1:8001 is now UP
Liste des serveurs down ["http://127.0.0.1:8000", "http://127.0.0.1:8001"]
Liste des serveurs down ["http://127.0.0.1:8000", "http://127.0.0.1:8001"]
Server http://127.0.0.1:8000 is now UP
Liste des serveurs up ["http://127.0.0.1:8000"]
Liste des serveurs down ["http://127.0.0.1:8001"]
Server http://127.0.0.1:8001 is now UP
Liste des serveurs up ["http://127.0.0.1:8000", "http://127.0.0.1:8001"]
Liste des serveurs up ["http://127.0.0.1:8000", "http://127.0.0.1:8001"]
```

Objectif 6 :

Ajout d'un menu pour choisir le port pour le proxy avec gestion d'erreur + création de fichier de log :

Pour le menu :

Il faut crée une loop dans le main avec une gestion d'erreur entre **1024-65534**

Pour les logs :

On va crée un fichier de log pour voir la répartition des requêtes entre nos différents serveur web, il faut crée une fonction **write_log()** qui crée un fichier *fxloadbalancer.log*, **on crée ce fichier de logs seulement quand l'utilisateur fait un ctrl+c.**

= txloadbalancer.log

```
1  Nombre total de requêtes : 61
2  Nombre de requêtes pour http://127.0.0.1:8000 : 43
3  Nombre de requêtes pour http://127.0.0.1:8001 : 18
4
```

```
use actix_web::{web, App, HttpResponse, HttpServer, http::Sta
tusCode, HttpRequest};
use awc::Client;
use rand::seq::SliceRandom;
use std::sync::{Mutex, Arc};
use lazy_static::lazy_static;
use std::time::Duration;
use actix_rt::time::interval;
use std::io::{self, Write};
use std::fs::File;
use std::collections::HashMap;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::process;
```

```
lazy_static! {
    static ref UPSTREAM_SERVERS: Mutex<Vec<(String, bool)>> =
Mutex::new(vec![
        ("http://127.0.0.1:8000".to_string(), true),
        ("http://127.0.0.1:8001".to_string(), true),
    ]);
    static ref REQUEST_COUNT: Arc<AtomicUsize> = Arc::new(Ato
micUsize::new(0));
    static ref SERVER_REQUESTS: Mutex<HashMap<String, usize>>
= Mutex::new(HashMap::new());
}
```

```
async fn try_upstream(server: &str, client: &Client, _req: &H
```

```

HttpRequest) -> Result<HttpResponse, ()> {
    let response = client.get(server).send().await;
    match response {
        Ok(resp) if resp.status().is_success() => {
            REQUEST_COUNT.fetch_add(1, Ordering::Relaxed);
            let mut server_requests = SERVER_REQUESTS.lock().
unwrap();
            *server_requests.entry(server.to_string()).or_inse
rt(0) += 1;
            Ok(HttpResponse::Ok().body(format!("Proxied to
{}", server)))
        }
        _ => Err(()),
    }
}

```

```

async fn redirect(req: HttpRequest) -> HttpResponse {
    let client = Client::default();
    let mut servers = UPSTREAM_SERVERS.lock().unwrap();

    let mut rng = rand::thread_rng();
    let servers_order: Vec<usize> = (0..servers.len()).collect();
    let mut servers_order_shuffled = servers_order.clone();
    servers_order_shuffled.shuffle(&mut rng);

    for index in servers_order_shuffled {
        let (server, alive) = &servers[index];
        if *alive {
            match try_upstream(server, &client, &req).await {
                Ok(response) => return response,
                Err(_) => servers[index].1 = false,
            }
        }
    }
}

```

```

        HttpResponse::new(StatusCode::SERVICE_UNAVAILABLE)
    }

    async fn check_upstream_servers() {
        let client = Client::default();
        let mut servers = UPSTREAM_SERVERS.lock().unwrap();
        for (server, alive) in servers.iter_mut() {
            let prev_alive = *alive;
            *alive = client.get(server.as_str()).send().await.is_
ok();

            if *alive != prev_alive {
                print_server_status(server, *alive);
            }
        }

        print_servers_summary(&servers);
    }

    fn print_server_status(server: &String, is_up: bool) {
        let (color_code, status) = if is_up { ("\x1b[32m", "UP")
} else { ("\x1b[31m", "DOWN") };
        println!("{}", "Server {} is now {}{}", color_code, server, s
tatus, "\x1b[0m");
    }

    fn print_servers_summary(servers: &Vec<(String, bool)>) {
        let up_servers: Vec<String> = servers.iter().filter(|(&_,
alive)| *alive).map(|(server, _)| server.clone()).collect();
        let down_servers: Vec<String> = servers.iter().filter(|
(&_, alive)| !*alive).map(|(server, _)| server.clone()).collec
t();

        if !up_servers.is_empty() {
            println!("{}", "Liste des serveurs up {:?}\x1b[0m",
up_servers);

```

```

    }
    if !down_servers.is_empty() {
        println!("\x1b[31mListe des serveurs down {:?}\x1b[0
m", down_servers);
    }
}

fn write_log() {
    let mut file = match File::create("fxloadbalancer.log") {
        Ok(file) => file,
        Err(err) => {
            eprintln!("Erreur lors de la création du fichier
de journal : {}", err);
            return;
        }
    };

    let requests = REQUEST_COUNT.load(Ordering::Relaxed);
    if let Err(err) = writeln!(&mut file, "Nombre total de re
quêtes : {}", requests) {
        eprintln!("Erreur lors de l'écriture dans le fichier
de journal : {}", err);
        return;
    }

    let server_requests = SERVER_REQUESTS.lock().unwrap();
    for (server, count) in server_requests.iter() {
        if let Err(err) = writeln!(&mut file, "Nombre de requ
êtes pour {} : {}", server, count) {
            eprintln!("Erreur lors de l'écriture dans le fich
ier de journal : {}", err);
            return;
        }
    }

    println!("Les informations de journal ont été écrites dan

```



```

s fxloadbalancer.log");
}

#[actix_web::main]
async fn main() -> io::Result<()> {
    let port: u16;
    loop {
        println!("Choisissez un port pour le bind (1024-65534) :");
        let mut port_input = String::new();
        io::stdin().read_line(&mut port_input).expect("Erreur de lecture de la ligne");
        port = match port_input.trim().parse() {
            Ok(num) if num >= 1024 && num <= 65534 => num,
            _ => {
                println!("Port invalide. Veuillez entrer un nombre entre 1024 et 65534.");
                continue;
            }
        };
        break;
    }

    let mut interval = interval(Duration::from_secs(10));
    actix_rt::spawn(async move {
        loop {
            interval.tick().await;
            check_upstream_servers().await;
        }
    });

    ctrlc::set_handler(move || {
        println!("\nArrêt du serveur...");
        write_log();
        process::exit(0);
    })
}

```

```
.expect("Erreur lors de la configuration du gestionnaire  
de signal");  
  
println!("Le serveur est en cours d'exécution. Appuyez su  
r Ctrl+C pour arrêter.");  
  
HttpServer::new(|| {  
    App::new().route("/", web::get().to(redirect))  
})  
    .bind(format!("127.0.0.1:{}", port))?  
    .run()  
    .await  
}
```

Objectif 7 :

Mettre en place l'algorithme de round-robin et enlever le random.

Round-robin ça fonctionne tour par tour implémenter cette logique !

Mise en place du round-robin :

Nous allons mettre en place l'une des méthodes les plus simples pour le load balancing et enlever notre version qui utilise random.

L'équilibrage de charge round-robin est l'une des méthodes les plus simples pour répartir les demandes des clients sur un groupe de serveurs. En parcourant la liste des serveurs du groupe, l'équilibreur de charge round-robin transmet une requête client à chaque serveur à tour de rôle. Lorsqu'il atteint la fin de la liste, l'équilibreur de charge revient en arrière et redescend la liste (il envoie la requête suivante au premier serveur de la liste, la suivante au deuxième serveur, et ainsi de suite).

```

use actix_web::{web, App, HttpResponse, HttpServer, http::Sta
tusCode, HttpRequest};
use awc::Client;
use std::sync::{Mutex, Arc};
use lazy_static::lazy_static;
use std::time::Duration;
use actix_rt::time::interval;
use std::io::{self, Write};
use std::fs::File;
use std::collections::HashMap;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::process;

lazy_static! {
    static ref UPSTREAM_SERVERS: Mutex<Vec<(String, bool)>> =
Mutex::new(vec![
        ("http://127.0.0.1:8000".to_string(), true),
        ("http://127.0.0.1:8001".to_string(), true),
    ]);
    static ref REQUEST_COUNT: Arc<AtomicUsize> = Arc::new(Ato
micUsize::new(0));
    static ref SERVER_REQUESTS: Mutex<HashMap<String, usize>>
= Mutex::new(HashMap::new());
    static ref ROUND_ROBIN_COUNTER: Arc<AtomicUsize> = Arc::n
ew(AtomicUsize::new(0));
}

async fn try_upstream(server: &str, client: &Client, _req: &H
ttpRequest) -> Result<HttpResponse, ()> {
    let response = client.get(server).send().await;
    match response {
        Ok(resp) if resp.status().is_success() => {
            REQUEST_COUNT.fetch_add(1, Ordering::Relaxed);
            let mut server_requests = SERVER_REQUESTS.lock().

```

```

unwrap();
        *server_requests.entry(server.to_string()).or_insert(0) += 1;
        Ok(HttpResponse::Ok().body(format!("Proxied to {}", server)))
    }
    _ => Err(()),
}

}

async fn redirect(req: HttpRequest) -> HttpResponse {
    let client = Client::default();
    let mut servers = UPSTREAM_SERVERS.lock().unwrap();
    let mut server_index = ROUND_ROBIN_COUNTER.fetch_add(1, Ordering::Relaxed) % servers.len();

    for _ in 0..servers.len() {
        let (server, alive) = &servers[server_index];
        if *alive {
            match try_upstream(server, &client, &req).await {
                Ok(response) => return response,
                Err(_) => servers[server_index].1 = false,
            }
        }
        server_index = (server_index + 1) % servers.len();
    }

    HttpResponse::new(StatusCode::SERVICE_UNAVAILABLE)
}

async fn check_upstream_servers() {
    let client = Client::default();
    let mut servers = UPSTREAM_SERVERS.lock().unwrap();
    let alive_servers_count = servers.iter().filter(|(_, alive)| *alive).count();

```

```

        if alive_servers_count == 1 {
            let alive_server_index = servers.iter().position(|(_,
alive)| *alive).unwrap();
            ROUND_ROBIN_COUNTER.store(alive_server_index as usiz
e, Ordering::Relaxed);
        }

        for (server, alive) in servers.iter_mut() {
            let prev_alive = *alive;
            *alive = client.get(server.as_str()).send().await.is_
ok();

            if *alive != prev_alive {
                print_server_status(server, *alive);
            }
        }

        print_servers_summary(&servers);
    }

fn print_server_status(server: &String, is_up: bool) {
    let (color_code, status) = if is_up { ("\x1b[32m", "UP")
} else { ("\x1b[31m", "DOWN") };
    println!("{}", "Server {} is now {}{}", color_code, server, s
tatus, "\x1b[0m");
}

fn print_servers_summary(servers: &Vec<(String, bool)>) {
    let up_servers: Vec<String> = servers.iter().filter(|(_,
alive)| *alive).map(|(server, _)| server.clone()).collect();
    let down_servers: Vec<String> = servers.iter().filter(|
(_, alive)| !*alive).map(|(server, _)| server.clone()).collec
t();

    if !up_servers.is_empty() {
        println!("{}", "\x1b[32mListe des serveurs up {:?}\x1b[0m",

```

```

up_servers);
    }
    if !down_servers.is_empty() {
        println!("\x1b[31mListe des serveurs down {:?}\x1b[0
m", down_servers);
    }
}

fn write_log() {
    let mut file = match File::create("fxloadbalancer.log") {
        Ok(file) => file,
        Err(err) => {
            eprintln!("Erreur lors de la création du fichier
de journal : {}", err);
            return;
        }
    };

    let requests = REQUEST_COUNT.load(Ordering::Relaxed);
    if let Err(err) = writeln!(&mut file, "Nombre total de re
quêtes : {}", requests) {
        eprintln!("Erreur lors de l'écriture dans le fichier
de journal : {}", err);
        return;
    }

    let server_requests = SERVER_REQUESTS.lock().unwrap();
    for (server, count) in server_requests.iter() {
        if let Err(err) = writeln!(&mut file, "Nombre de requ
êtes pour {} : {}", server, count) {
            eprintln!("Erreur lors de l'écriture dans le fich
ier de journal : {}", err);
            return;
        }
    }
}

```

```

    println!("Les informations de journal ont été écrites dans
fxloadbalancer.log");
}

#[actix_web::main]
async fn main() -> io::Result<()> {
    let port: u16;
    loop {
        println!("Choisissez un port pour le bind (1024-6553
4) :");
        let mut port_input = String::new();
        io::stdin().read_line(&mut port_input).expect("Erreur
de lecture de la ligne");
        port = match port_input.trim().parse() {
            Ok(num) if num >= 1024 && num <= 65534 => num,
            _ => {
                println!("Port invalide. Veuillez entrer un n
ombre entre 1024 et 65534.");
                continue;
            }
        };
        break;
    }

    let mut interval = interval(Duration::from_secs(10));
    actix_rt::spawn(async move {
        loop {
            interval.tick().await;
            check_upstream_servers().await;
        }
    });

    ctrlc::set_handler(move || {
        println!("\nArrêt du serveur...");
        write_log();
        process::exit(0);
    });
}

```

```

    })
    .expect("Erreur lors de la configuration du gestionnaire
de signal");

    println!("Le serveur est en cours d'exécution. Appuyez su
r Ctrl+C pour arrêter.");

    HttpServer::new(|| {
        App::new().route("/", web::get().to(redirect))
    })
    .bind(format!("127.0.0.1:{}", port))?
    .run()
    .await
}

```



ça fonctionne mais nous avons un problème... s'il y a qu'un seul serveur vivant le programme n'arrive plus à fonctionner, il faut donc gérer le cas où il n'y a qu'un seul serveur vivant dans le vecteur

```

use actix_web::{web, App, HttpResponse, HttpServer, http::Sta
tusCode, HttpRequest};
use awc::Client;
use std::{
    sync::{Mutex, Arc},
    time::Duration,
    io::{self, Write},
    fs::File,
    collections::HashMap,
    sync::atomic::{AtomicUsize, Ordering},
    process,
};

```



```

use lazy_static::lazy_static;
use tokio::time::interval;

lazy_static! {
    static ref UPSTREAM_SERVERS: Mutex<Vec<(String, bool)>> =
    Mutex::new(vec![
        ("http://127.0.0.1:8000".to_string(), true),
        ("http://127.0.0.1:8001".to_string(), true),
    ]);
    static ref REQUEST_COUNT: Arc<AtomicUsize> = Arc::new(AtomicUsize::new(0));
    static ref SERVER_REQUESTS: Mutex<HashMap<String, usize>> =
    Mutex::new(HashMap::new());
    static ref ROUND_ROBIN_COUNTER: Arc<AtomicUsize> = Arc::new(AtomicUsize::new(0));
}

async fn try_upstream(server: &str, client: &Client, _req: &HttpRequest) -> Result<HttpResponse, ()> {
    let response = client.get(server).send().await;
    match response {
        Ok(resp) if resp.status().is_success() => {
            REQUEST_COUNT.fetch_add(1, Ordering::Relaxed);
            let mut server_requests = SERVER_REQUESTS.lock().unwrap();
            *server_requests.entry(server.to_string()).or_insert(0) += 1;
            Ok(HttpResponse::Ok().body(format!("Proxied to {}", server)))
        }
        _ => Err(()),
    }
}

async fn redirect(req: HttpRequest) -> HttpResponse {
    let client = Client::default();

```

```

    let mut servers = UPSTREAM_SERVERS.lock().unwrap();
    let mut server_index = ROUND_ROBIN_COUNTER.fetch_add(1, Ordering::Relaxed) % servers.len();

    for _ in 0..servers.len() {
        let (server, alive) = &mut servers[server_index];
        if *alive {
            match try_upstream(server, &client, &req).await {
                Ok(response) => return response,
                Err(_) => *alive = false,
            }
        }
        server_index = (server_index + 1) % servers.len();
    }

    HttpResponse::new(StatusCode::SERVICE_UNAVAILABLE)
}

async fn check_upstream_servers() {
    let client = Client::default();
    let mut servers = UPSTREAM_SERVERS.lock().unwrap();
    let alive_servers_count = servers.iter().filter(|(_, alive)| *alive).count();

    if alive_servers_count == 1 {
        let alive_server_index = servers.iter().position(|(_, alive)| *alive).unwrap();
        ROUND_ROBIN_COUNTER.store(alive_server_index as usize, Ordering::Relaxed);
    }

    for (server, alive) in servers.iter_mut() {
        let prev_alive = *alive;
        *alive = client.get(server.as_str()).send().await.is_ok();
    }
}

```

```

        if *alive != prev_alive {
            print_server_status(server, *alive);
        }
    }

    print_servers_summary(&servers);
}

fn green(text: &str) -> String {
    format!("\x1b[32m{}\x1b[0m", text)
}

fn red(text: &str) -> String {
    format!("\x1b[31m{}\x1b[0m", text)
}

fn print_server_status(server: &String, is_up: bool) {
    let status = if is_up { green("UP") } else { red("DOWN") };
    println!("Server {} is now {}", server, status);
}

fn print_servers_summary(servers: &Vec<(String, bool)>) {
    let up_servers: Vec<String> = servers.iter().filter(|(_, alive)| *alive).map(|(server, _)| server.clone()).collect();
    let down_servers: Vec<String> = servers.iter().filter(|(_, alive)| !*alive).map(|(server, _)| server.clone()).collect();

    if !up_servers.is_empty() {
        print!("\rListe des serveurs up {}", green(format!("{:?}", up_servers).as_str()));
    } else {
        print!("\r");
    }
}

```

```

    }
    if !down_servers.is_empty() {
        print!("\rListe des serveurs down {}", red(format!("{}",
{:?}{}", down_servers).as_str())));
    } else {
        print!("\r");
    }
    io::stdout().flush().unwrap(); // Assure que la sortie es
t immédiatement affichée
}

fn write_log() {
    let mut file = match File::create("fxloadbalancer.log") {
        Ok(file) => file,
        Err(err) => {
            eprintln!("Erreur lors de la création du fichier
de journal : {}", err);
            return;
        }
    };

    let requests = REQUEST_COUNT.load(Ordering::Relaxed);
    if let Err(err) = writeln!(&mut file, "Nombre total de re
quêtes : {}", requests) {
        eprintln!("Erreur lors de l'écriture dans le fichier
de journal : {}", err);
        return;
    }

    let server_requests = SERVER_REQUESTS.lock().unwrap();
    for (server, count) in server_requests.iter() {
        if let Err(err) = writeln!(&mut file, "Nombre de requ
êtes pour {} : {}", server, count) {
            eprintln!("Erreur lors de l'écriture dans le fich

```

```

ier de journal : {}", err);
        return;
    }
}

println!("Les informations de journal ont été écrites dans
fxloadbalancer.log");
}

#[actix_web::main]
async fn main() -> io::Result<()> {
    let port: u16;
    loop {
        println!("Choisissez un port pour le bind (1024-6553
4) :");
        let mut port_input = String::new();
        io::stdin().read_line(&mut port_input).expect("Erreur
de lecture de la ligne");
        port = match port_input.trim().parse() {
            Ok(num) if num >= 1024 && num <= 65534 => num,
            _ => {
                println!("Port invalide. Veuillez entrer un n
ombre entre 1024 et 65534.");
                continue;
            }
        };
        break;
    }

    let mut interval = interval(Duration::from_secs(3));
    actix_rt::spawn(async move {
        loop {
            interval.tick().await;
            check_upstream_servers().await;
        }
    });
}

```

```

        ctrlc::set_handler(move || {
            println!("\nArrêt du serveur...");
            write_log();
            process::exit(0);
        })
        .expect("Erreur lors de la configuration du gestionnaire
de signal");

        println!("Le serveur est en cours d'exécution. Appuyez su
r Ctrl+C pour arrêter.");

        HttpServer::new(|| {
            App::new().route("/", web::get().to(redirect))
        })
        .bind(format!("127.0.0.1:{}", port))?
        .run()
        .await
    }
}

```

Objectif 8 :

Crée une méthode pour vérifier que si un client envoie plus de 10 requêtes en moins de 10 secondes sur le load balancer (peu importe le serveur qu'il vise) alors il se prend une erreur 429 directement par le load balancer, toutes les minutes se compteur est réinitialiser.

Je n'ai pas réussi à implémenter ce mécanisme ...

Objectif 9 :

Diviser le code en plusieurs fichiers, ajouter de la rustdocs, traduire le code en anglais, optimiser le code :

- colors.rs
- upstreams.rs

- main.rs
- server.rs
- logging.rs

main.rs

```
use std::{io, process};
use tokio::time::interval;
use actix_web::{web, App, HttpServer};
use std::time::Duration;

mod colors;
mod logging;
mod server;
mod upstream;

const MIN_PORT: u16 = 1024;
const MAX_PORT: u16 = 65534;

#[actix_web::main]
async fn main() -> io::Result<()> {
    let port: u16;
    loop {
        println!("Choose a port to bind ({}-{}) :", MIN_PORT,
MAX_PORT);
        let mut port_input = String::new();
        io::stdin().read_line(&mut port_input).expect("Error
reading input line");
        port = match port_input.trim().parse() {
            Ok(num) if num >= MIN_PORT && num <= MAX_PORT =>
num,
            _ => {
                println!("Invalid port. Please enter a number
between {} and {}.", MIN_PORT, MAX_PORT);
```

```

        continue;
    }
};
break;
}

// Set up a background task to periodically check upstream s
ervers.
let mut interval = interval(Duration::from_secs(3));
actix_rt::spawn(async move {
    loop {
        interval.tick().await;
        server::check_upstream_servers().await;
    }
});

// Register a Ctrl+C signal handler for graceful shutdown.
ctrlc::set_handler(move || {
    println!("\nServer shutting down...");
    logging::write_log();
    process::exit(0);
})
.expect("Error configuring signal handler");

println!("Server is running. Press Ctrl+C to stop");

// Start the Actix Web server to handle incoming request
s.
HttpServer::new(|| {
    App::new().route("/", web::get().to(server::redirec
t))
})
.bind(format!("127.0.0.1:{}", port))?
.run()

```



```
    .await  
}
```

upstream.rs

```
use std::sync::Mutex;  
  
lazy_static::lazy_static! {  
    /// Mutex-protected vector representing upstream servers  
    along with their availability status.  
    pub static ref UPSTREAM_SERVERS: Mutex<Vec<(String, bool)>  
> = Mutex::new(vec![  
        ("http://127.0.0.1:8000".to_string(), true),  
        ("http://127.0.0.1:8001".to_string(), true),  
    ]);  
}
```

server.rs

```
use actix_web::{HttpRequest, HttpResponse, http::StatusCode};  
use awc::Client;  
use std::sync::Arc;  
use std::sync::atomic::AtomicUsize;  
use crate::logging::REQUEST_COUNT;  
use crate::logging::SERVER_REQUESTS;  
use std::sync::atomic::Ordering;  
use crate::colors::{colorize, TextColor};  
use std::io;  
use std::io::Write;  
use crate::upstream::UPSTREAM_SERVERS;  
  
lazy_static::lazy_static! {  
    /// Atomic counter for round-robin load balancing.  
    static ref ROUND_ROBIN_COUNTER: Arc<AtomicUsize> = Arc::n  
ew(AtomicUsize::new(0));
```

```

}

pub async fn try_upstream(server: &str, client: &Client, _req: &HttpRequest) -> Result<HttpResponse, ()> {
    let response = client.get(server).send().await;
    match response {
        Ok(resp) if resp.status().is_success() => {
            REQUEST_COUNT.fetch_add(1, Ordering::Relaxed);
            let mut server_requests = SERVER_REQUESTS.lock().unwrap();
            *server_requests.entry(server.to_string()).or_insert(0) += 1;
            Ok(HttpResponse::Ok().body(format!("Proxied to {}", server)))
        }
        _ => Err(()),
    }
}

```

/// Redirects the incoming request to an available upstream server.

///

/// This function implements a round-robin load balancing strategy to redirect

/// incoming requests to available upstream servers. If all servers are down,

/// it returns a `Service Unavailable` response.

///

```

pub async fn redirect(req: HttpRequest) -> HttpResponse {
    let client = Client::default();
    let mut servers = UPSTREAM_SERVERS.lock().unwrap();
    let mut server_index = ROUND_ROBIN_COUNTER.fetch_add(1, Ordering::Relaxed) % servers.len();

```

```

    for _ in 0..servers.len() {
        let (server, alive) = &mut servers[server_index];

```

```

        if *alive {
            match try_upstream(server, &client, &req).await {
                Ok(response) => return response,
                Err(_) => *alive = false,
            }
        }
        server_index = (server_index + 1) % servers.len();
    }

    HttpResponse::new(StatusCode::SERVICE_UNAVAILABLE)
}

/// Checks the availability of upstream servers.
///
/// This function sends a health check request to each upstream
/// server to determine
/// its availability. It updates the status of each server and
/// prints a summary of
/// the available and unavailable servers to the console.
pub async fn check_upstream_servers() {
    let client = Client::default();
    let mut servers = UPSTREAM_SERVERS.lock().unwrap();
    let alive_servers_count = servers.iter().filter(|(_, alive)| *alive).count();

    if alive_servers_count == 1 {
        let alive_server_index = servers.iter().position(|(_, alive)| *alive).unwrap();
        ROUND_ROBIN_COUNTER.store(alive_server_index as usize, Ordering::Relaxed);
    }

    for (server, alive) in servers.iter_mut() {
        let prev_alive = *alive;
        *alive = client.get(server.as_str()).send().await.is_ok();
    }
}

```

```

        if *alive != prev_alive {
            print_server_status(server, *alive);
        }
    }

    print_servers_summary(&servers);
}

fn print_server_status(server: &String, is_up: bool) {
    let status = if is_up { colorize("UP", TextColor::Green)
} else { colorize("DOWN", TextColor::Red) };
    println!("Server {} is now {}", server, status);
}

/// * `servers` - A reference to a vector containing tuples o
f server URLs and their availability status.
fn print_servers_summary(servers: &Vec<(String, bool)>) {
    let up_servers: Vec<String> = servers.iter().filter(|(_,
alive)| *alive).map(|(server, _)| server.clone()).collect();
    let down_servers: Vec<String> = servers.iter().filter(|
(_, alive)| !*alive).map(|(server, _)| server.clone()).collec
t();

    if !up_servers.is_empty() {
        print!("\rListe des serveurs up {}", colorize(&forma
t!("{:?}", up_servers), TextColor::Green));
    } else {
        print!("\r");
    }
    if !down_servers.is_empty() {
        print!("\rListe des serveurs down {}", colorize(&form
at!("{:?}", down_servers), TextColor::Red));
    } else {
        print!("\r");
    }
    io::stdout().flush().unwrap(); // Ensures immediate output

```

```
t  
}
```

logging.rs

```
use std::{  
    fs::File,  
    io::Write,  
    sync::{Arc, Mutex},  
    collections::HashMap,  
};  
use std::sync::atomic::AtomicUsize;  
use std::sync::atomic::Ordering;  
  
use lazy_static::lazy_static;  
  
lazy_static! {  
    /// Atomic counter to track the total number of requests received by the server.  
    pub static ref REQUEST_COUNT: Arc<AtomicUsize> = Arc::new(AtomicUsize::new(0));  
    /// Mutex-protected map to track the number of requests received by each server.  
    pub static ref SERVER_REQUESTS: Mutex<HashMap<String, usize>> = Mutex::new(HashMap::new());  
}  
  
/// Function to write server request statistics to a log file  
pub fn write_log() {  
    let mut file = File::create("fxloadbalancer.log")  
        .unwrap_or_else(|err| panic!("Error creating log file: {}", err));  
}
```

```

    let requests = REQUEST_COUNT.load(Ordering::Relaxed);

    writeln!(&mut file, "Total number of requests : {}", requests)
        .unwrap_or_else(|err| eprintln!("Error writing to log file : {}", err));

    let server_requests = SERVER_REQUESTS.lock().unwrap();
    for (server, count) in server_requests.iter() {
        writeln!(&mut file, "Number of requests for {} : {}", server, count)
            .unwrap_or_else(|err| eprintln!("Error writing to log file : {}", err));
    }

    println!("Log information has been written to fxloadbalancer.log");
}

```

colors.rs

```

const RESET_COLOR: &str = "\x1b[0m";
const GREEN_COLOR: &str = "\x1b[32m";
const RED_COLOR: &str = "\x1b[31m";

pub enum TextColor {
    Green,
    Red,
}

/// Function to colorize text with specified color
pub fn colorize(text: &str, color: TextColor) -> String {
    match color {
        TextColor::Green => format!("{}", GREEN_COLOR, text, RESET_COLOR),
    }
}

```

```
        TextColor::Red => format!("{}", RED_COLOR, text,  
RESET_COLOR),  
    }  
}
```