CONFIGURAR UN SERVIDOR EN RUST DE MANERA PROFESIONAL

1. Configurar caché, código 304 y ETag

Para archivos estáticos (CSS, JS, etc.), puedes agregar encabezados de caché y calcular un ETag para re

```
use actix_web::{middleware, HttpResponse};

fn file_handler() -> HttpResponse {
    HttpResponse::Ok()
        .append_header(("Cache-Control", "max-age=31536000")) // 1 año
        .append_header(("ETag", "custom-etag-value"))
        .body("Contenido del archivo estático")
}
```

Actix-Web también tiene soporte para servir archivos estáticos directamente usando actix_files:

```
use actix_files::Files;

App::new()
    .service(Files::new("/static", "./static")
        .prefer_utf8(true)
        .use_etag(true)
        .use_last_modified(true))
```

2. Manejador global de errores (500, página personalizada)

Puedes configurar un middleware global para capturar errores y devolver páginas personalizadas:

```
use actix_web::{dev::ServiceRequest, dev::ServiceResponse, http::StatusCode, Error, middleware, App, H

async fn error_handler<B>(res: ServiceResponse<B>) -> Result<middleware::ErrorHandlerResponse<B>>
    let response = match res.response().status() {
        StatusCode::INTERNAL_SERVER_ERROR => HttpResponse::InternalServerError()
            .body("Página personalizada de error 500"),
        _ => res.into_response(),
    };

    Ok(middleware::ErrorHandlerResponse::Response(response))
}

// Añadir el middleware:
App::new()
    .wrap(middleware::ErrorHandlers::new().handler(StatusCode::INTERNAL_SERVER_ERROR, error_har
```

3. Configurar errores 400 (Bad Request, 404, etc.)

Puedes manejar errores como 400, 404, etc., usando un middleware o una configuración por ruta:

```
App::new()
    .default_service(
        web::route().to(|| HttpResponse::NotFound().body("Página no encontrada")),
    )
```

4. Redirecciones 301 y 302

Usa las respuestas apropiadas para redirecciones:

```
use actix_web::{HttpResponse, web};

async fn redirect_301() -> HttpResponse {
    HttpResponse::MovedPermanently()
        .append_header(("Location", "/nuevo-destino"))
        .finish()
}

async fn redirect_302() -> HttpResponse {
    HttpResponse::Found()
        .append_header(("Location", "/temporal-destino"))
        .finish()
}

App::new()
    .route("/antigua-url", web::get().to(redirect_301))
    .route("/temporal-url", web::get().to(redirect_302))
```

5. Configurar longitud máxima de URL

Puedes controlar el tamaño máximo de la URL y devolver errores apropiados:

```
.use(middleware::NormalizePath::new(middleware::TrailingSlash::Trim))
.use(middleware::ErrorHandlers::new().handler(StatusCode::REQUEST_URI_TOO_LONG, |_, _| {
    Ok(HttpResponse::UriTooLong().finish())
}))
```

6. Configurar tamaño máximo de petición y headers

Define el tamaño máximo permitido para las solicitudes o archivos subidos:

```
App::new()
    .app_data(web::PayloadConfig::default().limit(10 * 1024 * 1024)) // Tamaño máximo 10MB
    .wrap(middleware::DefaultHeaders::new().add(("Max-Upload-Size", "10MB")))
```

7. Configurar conexiones, threads y monitorización

Conexiones máximas y workers:

```
HttpServer::new(|| {
    App::new()
        .route("/", web::get().to(index_page))
})
.workers(8)
.max_connections(10_000)
.max_connection_rate(1_000)
```

Monitorización:

No hay soporte nativo, pero puedes integrar Prometheus usando actix-web-prom para métricas en tiempo

8. Configurar respuesta HEADER por URL

Usa DefaultHeaders para añadir encabezados globales o específicos:

```
App::new()
    .wrap(middleware::DefaultHeaders::new().add(("X-Custom-Header", "Value")))
```

9. Configurar CORS, HSTS, y protección contra ataques

CORS:

```
use actix_cors::Cors;

App::new()
    .wrap(
        Cors::default()
            .allowed_origin("https://example.com")
            .allowed_methods(vec!["GET", "POST"])
            .allowed_headers(vec![http::header::CONTENT_TYPE])
            .max_age(3600),
    )
```

HSTS y Clickjacking:

```
.wrap(middleware::DefaultHeaders::new()
    .add(("Strict-Transport-Security", "max-age=63072000; includeSubDomains"))
    .add(("X-Frame-Options", "DENY"))
    .add(("X-Content-Type-Options", "nosniff"))
)
```

10. Protección contra ataques DOS y errores 429

Limita conexiones por cliente y la tasa de solicitudes. Puedes usar middleware externo o implementar algo

```
use std::sync::Arc;
use std::collections::HashMap;
```

```
use tokio::sync::Mutex;
use actix_web::{HttpServer, App, HttpResponse, web};

type RateLimiter = Arc<Mutex<HashMap<String, u32>>>;

async fn rate_limit(limiter: web::Data<RateLimiter>, ip: String) -> HttpResponse {
    let mut limiter = limiter.lock().await;
    let counter = limiter.entry(ip).or_insert(0);
    if *counter >= 100 {
        HttpResponse::TooManyRequests().finish()
    } else {
        *counter += 1;
        HttpResponse::Ok().body("Acceso permitido")
    }
}
```

11. Configurar timeout de conexiones

Puedes ajustar los timeouts de conexiones usando el HttpServer:

```
HttpServer::new(|| {
    App::new().route("/", web::get().to(index_page))
})
.client_timeout(std::time::Duration::from_secs(30)) // Timeout por cliente
.client_shutdown(std::time::Duration::from_secs(5)) // Tiempo para cerrar conexiones
```

Resumen:

Este enfoque configura tu servidor Rust de manera profesional, abarcando:

1. Seguridad (CORS, HSTS, XSS, etc.).

2. Optimización (caché, conexiones, workers).

3. Manejabilidad (errores personalizados, métricas).