Ejan Shrestha

A working portfolio

Systems that stay up

I build the .NET backends that insurance and payments in Nepal run on, and keep them running in production.

Ejan Shrestha इजन श्रेष्ठ

Senior Software Engineer, Kathmandu, Nepal

Kathmandu, 2026 edition

Contents

Foreword

Who I am and what I do, on one page.

I Projects

11 things I've built, one chapter each.

II Deep dives

6 problems from the projects, in depth.

III Experience

Where I've worked, my skills and education.

IV Notes from production

All 14 articles, in full.

V Places

Where I've lived and worked.

Kathmandu, 2026 edition

The living version, with the interactive architecture map and globe, is at ejanshrestha.com.np.

Set in Bricolage Grotesque, Geist and Geist Mono.

Foreword

I build the .NET backends that insurance and payments in Nepal run on, and keep them running in production. Right now that's a 50-project insurance platform and the portals its customers buy and claim through.

I'm a senior software engineer in Kathmandu with 4+ years shipping .NET for insurance, payments and enterprise. At Himalayan Everest Insurance I look after a platform of 50 .NET projects on Kubernetes: I rebuilt nine of its services on .NET 8, ended socket exhaustion across 397 HttpClient call sites, and wrote most of the API behind its online portals. Before that, at EKbana, I built a digital wallet's agent platform and designed its partner API. I write about what I learn, and most of it started as an incident.

.NET projects	services rebuilt on .NET 8	HttpClient call sites fixed	portal API endpoints
50	9	397	298

Looking for

Open to senior and lead roles in backend architecture and .NET modernisation, in Kathmandu or remote.

Reach me

sthaejan00@gmail.com
+977 986 213 5998
linkedin/ in/ejanshrestha
github/ ijanggmu
x/ @stha_ejan
hashnode/ ijanggmu.hashnode.dev

Part I

Projects

11 things I've built or run, from the insurance platform I look after at work to my own products.

Chapter 01

A 50-project insurance platform

2022 - present / Vendor engineer, then senior engineer in-house

9

services rebuilt on .NET 8

Problem

Policy, claims, accounting, reinsurance and reporting services, most on end-of-life .NET, each logging its own way and failing under load.

What I did

Built co-insurance accounting and receipts as a vendor engineer. In-house, rebuilt nine services on .NET 8, moved Hangfire to .NET 8, replaced 397 per-call HttpClients with IHttpClientFactory, and put every service on one Serilog and Seq library with correlation IDs.

Result

The platform went from zero to thirteen .NET 8 projects without stopping, and outbound HTTP stopped exhausting sockets.

Shape of the system



Stack

.NET Core 2.2 → .NET 8, PostgreSQL, Hangfire, IdentityServer4, Serilog + Seq, Docker, Kubernetes

Chapter 02

An insurer's online portals

2024 - present / Main backend author; started the admin console

298

endpoints across eight portal APIs, most of them mine

Problem

Customers had to visit a branch to buy, renew or claim, and agents, corporate clients and partners had no way in at all.

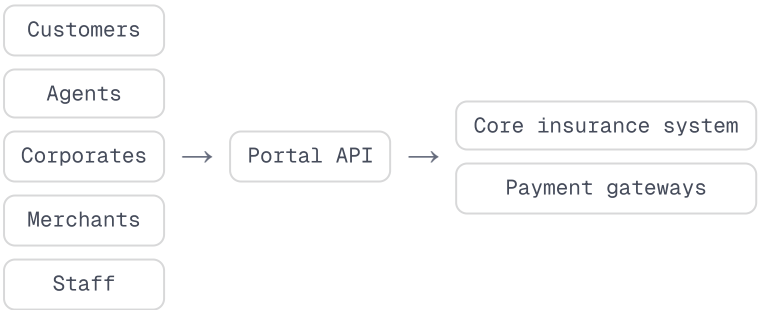
What I did

Built the API behind the customer, agent, corporate, merchant and admin portals: KYC, 2FA, quotes, purchase, renewals, claims and policy PDFs. eSewa, Khalti and ConnectIPS sit behind one payment interface, a typed client covers 62 core-system endpoints, and partners use a merchant API with HMAC signing and idempotency keys.

Result

Customers, agents and partners buy, renew and claim online, and staff run it from one permissioned console.

Shape of the system



Stack

.NET 8, PostgreSQL, Hangfire, SignalR, Refit, Serilog + Seq, Next.js 16

Chapter 03

Reports that don't slow the platform down

2022 - present / Backend engineer, then senior engineer

goAML

anti-money-laundering reports, generated in Postgres

Problem

Heavy reports ran on the databases that issue policies, data was split across one database per service, and the busiest tables kept growing.

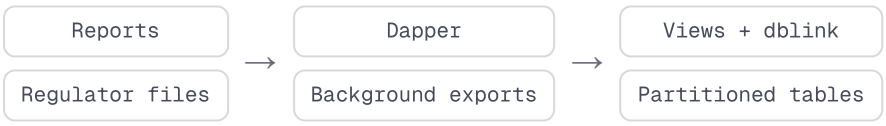
What I did

Generated goAML anti-money-laundering reports from a Postgres function read with Dapper. Joined service databases with views and dblink, partitioned the high-volume tables, kept multi-service writes consistent, and moved big exports to a Hangfire queue that streams to MinIO.

Result

Reports and regulator files come out without slowing policy issuance, and large tables stay fast as they grow.

Shape of the system



Stack

PostgreSQL, Views, dblink, Partitioning, Dapper, Hangfire, MinIO

Chapter 04

Keeping the cluster running

2025 - present / Hands-on on the production servers

98

SocketException 98, traced from a pod back to the code

Problem

The platform runs as dozens of workloads on Kubernetes, and when one misbehaves in production the cause is rarely in the code you're looking at.

What I did

Roll out and roll back services, image tags, config maps and secrets. Debug production pods from logs and shells, as when SocketException 98 in the underwriting pod traced back to exhausted ports. Manage ingress, TLS certificates and service routing, and set resources, replicas, probes and persistent volumes.

Result

Releases and incidents are handled on the cluster itself, not escalated and waited on.

Shape of the system



Stack

Kubernetes, kubectl, nginx ingress, cert-manager, Docker, Persistent volumes

Chapter 05

A digital wallet and agent network

2023 - 2025 / Backend engineer and reviewer, at EKbana

2,119

of my commits on its core API, 165 teammates' merge requests reviewed

Problem

A wallet needed customers, agents and staff on one platform for KYC, transfers, remittance and bill payments, without paying a remittance out twice.

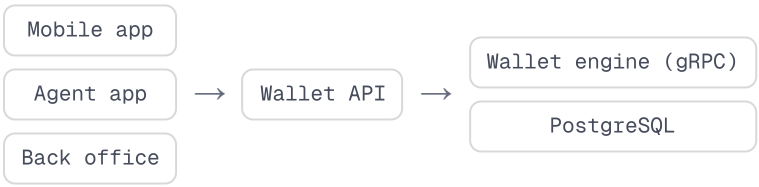
What I did

Built agent, branch, customer and KYC modules, OTP, lockouts and PDF and Excel statements. Stopped double payouts with a five-minute claim lock and an auto-unlock job, integrated AML screening, and reviewed and merged teammates' work into qa, uat and live.

Result

One API family serves three apps through 238 gRPC calls to the wallet engine, and a remittance can only be paid once.

Shape of the system



Stack

.NET 7, gRPC, PostgreSQL, EF Core, Hangfire, Firebase, MinIO, Jenkins

Chapter 06

A partner API I designed for a wallet

2023 - 2024 / Designed and bootstrapped it; the team built features on it

53

partner endpoints on the structure I set up

Problem

Payment partners needed to pay out remittances, validate customers and push payments into the wallet without touching the core apps.

What I did

Started the service and wrote 94% of its first code: partner auth with IP allow-lists, gRPC to the wallet engine, Dapper reads of the core database, audit logs, permissions and a two-tier per-IP rate limiter. Pay tokens are single-use, expire, and can't be issued twice.

Result

Partners integrate through one audited API, and the team added payout, utility and QR features on its structure.

Shape of the system



Stack

.NET 7 → 8, gRPC, Dapper, PostgreSQL, Rate limiting, Serilog, Docker

Chapter 07

Accounting data out of QuickBooks

2024 / Main backend developer, at EKbana

16

kinds of accounting record imported

Problem

An accounting firm's clients kept their books in QuickBooks Online, and staff rebuilt trial balances and ledgers by hand in spreadsheets.

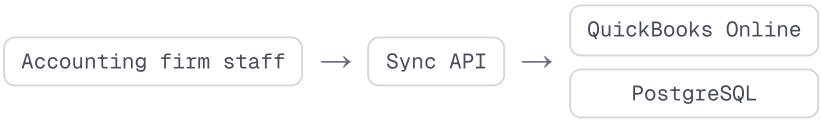
What I did

Built the QuickBooks OAuth connection with encrypted tokens and a scheduled refresh job, imported sixteen kinds of accounting record with bulk inserts, and built the trial balance and general ledger with an accounting-method switch, saved column choices and Excel export.

Result

Staff connect a client once and pull ready reports, instead of retyping them.

Shape of the system



Stack

.NET 8, QuickBooks API, OAuth 2.0, PostgreSQL, Hangfire, EPPlus, Jenkins

Chapter 08

An insurer's website its team edits in two languages

2026 / Lead developer: site, CMS and deployment, with a junior developer

2

languages, edited from one admin

Problem

Content changes on an insurer's website needed a developer, and Nepali pages lagged behind the English ones.

What I did

Built a Next.js site on Strapi 5 in English and Nepali: product pages with lead forms, careers with CV uploads and bulk export, news and insights. Deployed with Docker behind nginx, wrote the incident runbooks, and reviewed the junior developer's work.

Result

The marketing team publishes in both languages from one admin, without a developer.

Shape of the system



Stack

Next.js 15, React 19, Strapi 5, PostgreSQL, i18n, Docker, nginx

Chapter 09

Book a court from your phone

2025 - 2026 / Lead backend developer, with three contributors

237

endpoints across 18 booking modules

Problem

Players booked futsal courts by phone, venues tracked slots by hand, and two people could pay for the same hour.

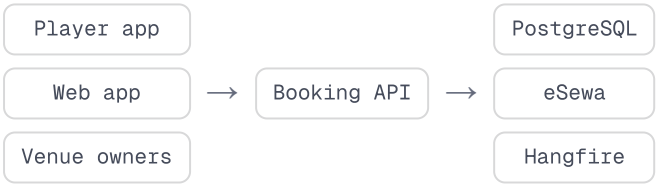
What I did

Built the booking API: slots are held for a limited time while you pay and released by a background job, pending payments expire, and eSewa pays each venue owner directly with the venue's own credentials. Added recurring bookings, pricing rules, wallets, loyalty points, teams, tournaments, organiser sign-up with OTP and Google sign-in.

Result

One API serves the player app, the web app and the venue owners, and a slot can't be sold twice.

Shape of the system



Stack

.NET 8, PostgreSQL, Hangfire, MinIO, eSewa, Next.js 16

Chapter 10

A pharmacy system built for Nepal

1,900
tests in the modules I wrote

2025 - 2026 / Solo, on the open-source fullstackhero kit, paired with AI

Problem

Nepali pharmacies juggle stock, VAT, tax deductions and IRD paperwork across separate tools.

What I did

Added thirteen modules to a .NET 10 modular monolith, from accounting and inventory to payroll and prescriptions: IRD CBMS e-billing and VAT returns, TDS and SSF payroll, Fonepay QR, weighted-average batch costing, a narcotics register and Nepali invoices.

Result

One multi-tenant system covers a pharmacy from purchase to tax return, backed by a large test suite.

Shape of the system



Stack

.NET 10, Modular monolith, PostgreSQL, Redis, Next.js 16, React 19, Aspire

Chapter 11

A life and money dashboard my AI can read

2026 / Solo build

MCP

server, so AI assistants can query my portfolio

Problem

Expenses, habits, loans and a NEPSE share portfolio lived in separate spreadsheets that nothing could reason over.

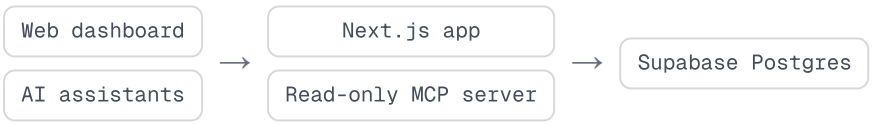
What I did

Built one dashboard for budgets, habits, journal, health, loans and investments, with candlestick charts, price-history backfill and gold and silver valuation. Added an AI analyst and a daily market brief on the Anthropic SDK, and a read-only MCP server secured with hashed personal tokens so assistants can query the portfolio.

Result

I ask an AI assistant about my own money and it answers from live data, read-only.

Shape of the system



Stack

Next.js 16, React 19, Supabase, Anthropic SDK, MCP, Recharts

Part II

Deep dives

Six problems from the projects, written the way I'd brief a new engineer: what broke, what I changed, what it bought us.

Case 01

397

new HttpClient() call sites retired

Insurance platform

Ending socket exhaustion across the platform

Problem

Under load, outbound calls failed with address-already-in-use errors, and the underwriting pod's database connections broke with them.

What I did

Traced it to `new HttpClient()` created on every call and left in TIME_WAIT. Replaced 397 call sites in 123 files with IHttpClientFactory named and typed clients, and gave every client the correlation-ID handler.

Result

Occurrences went from 414 to 17, all of them intentional or commented out, and the error class left production logs.

Case 02

1 ID

follows a request across every service

Insurance platform

One trace from gateway to database

Problem

Every service carried its own copy of logging code, and the .NET Core 2.2 services logged almost nothing useful.

What I did

Moved Serilog, Seq and correlation-ID code into one shared library, built a mirror of it for the 2.2 services with global exception middleware, and put the partner API and IdentityServer on it.

Result

Every response carries an X-Correlation-ID, one ID follows a request end to end in Seq, and errors return a safe body instead of internals.

Case 03

0 → 13

.NET 8 projects in the platform

Insurance platform

From end-of-life .NET to .NET 8

Problem

When I joined in-house, 36 of the platform's 37 projects ran on .NET Core 2.2, 3.1 or 5, all out of support.

What I did

Built new .NET 8 versions of nine services, among them accounting, claims, reinsurance, CRM and underwriting, moved Hangfire to .NET 8, and started the partner API's move to .NET 10.

Result

The platform went from zero to thirteen .NET 8 projects, and accounting, claims and reinsurance features now ship on the new services.

Case 04

5 min

claim lock on every remittance payout

Digital wallet

Paying a remittance out only once

Problem

Two agents could look up and pay out the same cash-pickup remittance at the same time, and the money would leave twice.

What I did

Gave every remittance a five-minute claim: the first agent to track it locks it and others are turned away, a background job releases stale locks, and back-office staff can unlock one by hand. Partner payouts use single-use pay tokens that expire.

Result

A remittance can be claimed by one agent at a time and paid out only once.

Case 05

1×

job registration at startup, not per request

Insurance platform

Background jobs that run once

Problem

Recurring jobs were scheduled from request handling, so every request re-registered all of them, and full-year report exports timed out inside the web request.

What I did

Moved scheduling into a hosted service that runs once at startup. Built background exports on a dedicated Hangfire queue with concurrency caps, streaming rows to Excel or CSV and storing files in MinIO for seven days.

Result

Jobs register once, and large reports finish in the background instead of timing out.

Case 06

3

payment gateways behind one interface

Online portals

Payments partners can't send twice

Problem

The portals needed eSewa, Khalti and ConnectIPS, and partners retrying a request could have created the same policy twice.

What I did

Put the three gateways behind one payment interface, and gave the merchant API HMAC request signing and idempotency keys stored per partner, method, path and key, with expiry.

Result

Adding a gateway doesn't touch purchase code, and a retried partner request returns the first result instead of acting again.

Part III

Experience

From first commit to platform owner.

Senior Software Engineer, Himalayan Everest Insurance

Jan 2025 - present

Kathmandu, hybrid

- Own backend reliability and modernisation for an insurance platform of 50 .NET projects on Kubernetes, across policy, claims, accounting, reinsurance and reporting.
- Rebuilt nine services on .NET 8 and moved Hangfire to .NET 8; replaced 397 per-call HttpClients with IHttpClientFactory to end socket exhaustion.
- Main backend author of the online portals: 298 endpoints for customers, agents, corporates, merchants and staff, with eSewa, Khalti and ConnectIPS payments.
- Built one Serilog and Seq logging library with correlation IDs; reporting on PostgreSQL views, dblink and partitioned tables; hands-on with the production Kubernetes cluster.
- Integrate and review the team's work: about 320 teammates' branches merged from some 13 developers since 2025.

.NET Developer, EKbana Solutions

Jul 2022 - Nov 2024

Lalitpur, on-site

- Vendor engineer on the insurance platform: co-insurance accounting, receipts, trial balance optimisation, and goAML anti-money-laundering reports built in PostgreSQL and read with Dapper.
- Backend engineer on a digital wallet and agent network (2,119 commits); designed its partner API and stopped double remittance payouts with a claim lock.
- Reviewed and merged 165 merge requests from seven teammates, and promoted releases through qa, uat and live.
- Built a QuickBooks Online integration for an accounting firm, and the base API (auth, permissions, Hangfire, audit and request logging, rate limiting) later used for the insurer's portals.

.NET Developer, NBIT

Jan 2022 - Jun 2022

Lalitpur

- Built and maintained .NET Framework applications for client projects.

Skills

Backend	C#, ASP.NET Core, EF Core, Dapper, Hangfire, gRPC, SignalR, Refit, IdentityServer, YARP
Data	PostgreSQL, Views and dblink, Partitioning, SQL Server, Redis, MinIO, Query tuning, Migrations
Platform	Docker, Kubernetes, Nginx, Traefik, Linux, Let's Encrypt
Observability	Serilog, Seq, Grafana Loki, Correlation IDs
Front end	Next.js, React, TypeScript, Tailwind, Strapi

Education

BSc Computer Science and Information Technology
Vedas College, 2017 – 2022

Part IV

Notes from production

14 articles on .NET, infrastructure and running a team. Most started as an incident.

The articles

01	Git branching and release standard	15 Sept 2026
02	Deploying a new service in UAT with Docker and Nginx	22 Feb 2026
03	Handling forwarded headers in ASP.NET Core behind Nginx	26 Jun 2025
04	Defending a .NET API against DDoS attacks	12 Jun 2025
05	Documenting a system you inherited	19 Nov 2024
06	Nginx and free SSL for a .NET app with Certbot	5 Nov 2024
07	Mastering HttpClient in .NET with Refit	21 Aug 2024
08	Triggers in Postgres	20 Aug 2024
09	Exploring YARP, the reverse proxy for .NET	19 Aug 2024
10	Cancellation tokens: a practical guide	15 Aug 2024
11	Rate limiting in .NET, from concept to code	15 Aug 2024
12	Popular design patterns in .NET	5 May 2024
13	Dynamic objects in C#	8 Mar 2023
14	ViewBag, ViewData and TempData in ASP.NET MVC	8 Mar 2023

Article 01 / 15 Sept 2026 / Process / 3 min read

Git branching and release standard

The branch structure and promotion path my team ships with across dev, UAT and prod, written as a standard anyone can follow.

Version: 1.0 **Applies to:** All development teams and repositories **Environments:** DEV, UAT, PROD

1. Branch Structure

We use the following permanent branches:

```
main
dev
uat
prod
```

Temporary branches:

```
feature/*
bugfix/*
release/*
hotfix/*
```

Branch purpose

Branch	Purpose	Environment
`main`	Stable integration/source-of-truth branch	—
`dev`	Active development	DEV
`uat`	UAT-approved release candidate	UAT
`prod`	Production code	PROD
`feature/*`	New functionality	—
`bugfix/*`	Normal bug fixes	—
`release/*`	Release stabilization	—
`hotfix/*`	Critical production fixes	—

2. Standard Development Flow

```
feature/*
     ↓
    PR
     ↓
   dev
     ↓
    DEV
     ↓
release/*
     ↓
    UAT
     ↓
   UAT
     ↓
   prod
     ↓
   PROD
```

Example

```
feature/payment-integration
        ↓
       PR
        ↓
       dev
        ↓
       DEV
        ↓
release/2.5.0
        ↓
       UAT
        ↓
       prod
        ↓
      PROD
```

3. Feature Branches

All new development must start from `dev`.

```
git checkout dev
git pull origin dev

git checkout -b feature/customer-payment
```

Naming:

```
feature/customer-payment
feature/google-login
feature/claim-search
```

Keep feature branches short-lived.

Do not directly commit to `dev`, `uat`, or `prod`.

4. Bug Fixes

Normal bugs:

```
bugfix/claim-calculation
bugfix/payment-callback
bugfix/report-filter
```

Create from `dev`:

```
dev
 ↓
bugfix/claim-calculation
 ↓
PR
 ↓
dev
```

5. Release Branch

When the development version is ready for UAT:

```
dev
 ↓
release/2.5.0
```

After creating a release branch:

- No new features
- Only bug fixes
- Release/configuration fixes are allowed
- Changes require PR/review

The release branch is deployed to UAT.

```
release/2.5.0
       ↓
      UAT
```

6. UAT → Production

After UAT approval:

```
release/2.5.0
      ↓
     prod
      ↓
    PROD
```

Create a Git tag:

```
v2.5.0
```

Production must always be traceable to a specific commit/tag.

---

7. Production Hotfix

For critical production issues:

```
prod
 ↓
hotfix/payment-failure
 ↓
PR
 ↓
prod
 ↓
PROD
```

After the hotfix is released, the fix **must be merged/back-ported into** `dev` and any active release branch that contains the affected code.

```
        hotfix
       /   |   \
      ↓    ↓    ↓
   prod   uat   dev
```

This prevents the production fix from being lost in future releases.

---

8. Pull Request Rules

Every change must go through a Pull Request.

Minimum requirements:

- CI/build must pass
- Tests must pass
- Code review required
- No unresolved comments
- No merge conflicts
- Branch must be up to date where required

Production branches must be protected.

```
❌ Direct push ⟶ prod
❌ Direct push ⟶ uat
❌ Direct push ⟶ dev

✅ Branch ⟶ PR ⟶ Review ⟶ Merge
```

---

9. Environment Rules

DEV

```
dev ⟶ DEV
```

Automatic deployment is allowed.

UAT

```
release/* ⟶ UAT
```

Deployment should be controlled and associated with a specific release.

PROD

```
prod ⟶ PROD
```

Production deployment requires explicit approval.

---

10. Docker / CI-CD Rule

Build once, deploy the same artifact.

Do not rebuild the application separately for DEV, UAT, and PROD.

```
Source Code
    ↓
Build
    ↓
Docker Image
    ↓
GHCR
    ↓
DEV
    ↓
UAT
    ↓
PROD
```

Example:

```
ghcr.io/company/claim-api:2.5.0
```

Prefer immutable image tags/digests for production.

Avoid relying on:

```
:latest
```

for production deployments.

---

11. Database Migration Rule

Database migrations must be:

1. Created and tested in DEV
2. Tested in UAT

3. Reviewed before PROD
4. Executed in a controlled deployment process

Avoid automatically running destructive database migrations during application startup in PROD.

---

12. Branch Naming Standard

Use lowercase and descriptive names.

```
feature/customer-login
feature/esewa-payment

bugfix/claim-calculation
bugfix/report-filter

release/2.5.0

hotfix/payment-timeout
hotfix/claim-production-error
```

Avoid:

```
test
new-feature
fix
my-branch
ejan-work
temp
final
final2
```

---

13. Golden Rules

Every developer should remember these **10 rules**:

1. **Never push directly to `prod`.**
2. **Never push directly to `uat`.**
3. **Use a feature/bugfix branch for development.**
4. **All changes go through PR and code review.**
5. **`dev` is for active development.**
6. **`release/*` is for UAT stabilization.**
7. **`uat` contains the release being validated.**
8. **`prod` contains production code.**
9. **Every production release gets a version tag.**
10. **Production/UAT fixes must be back-merged into `dev`.**

One-line workflow

```
FEATURE ⟶ DEV ⟶ RELEASE ⟶ UAT ⟶ PROD
```

Emergency workflow

```
PROD ⟶ HOTFIX ⟶ PROD
          ↓
      back-merge ⟶ DEV
```

This is the standard I would put into your team's engineering documentation.

Article 02 / 22 Feb 2026 / Infrastructure / 2 min read

Deploying a new service in UAT with Docker and Nginx

A checklist for adding a service to a shared UAT server without breaking the ones already running there.

This guide provides a step-by-step approach to deploy a new service in the UAT environment using Docker, configure it in Nginx, and secure it with SSL.

1. Create a Domain

1. Choose a domain or subdomain for the new service, for example:

newservice.uat.example.com

1. Add an A or CNAME record in your DNS provider pointing the domain to your UAT server's public IP.

2. Configure Nginx

1. Open the UAT Nginx configuration file:

sudo nano /home/ubuntu/project/hei-core-new/nginx.uat.conf

1. Add a new server block for the service:

```
server {
  listen 443 ssl http2;
  server_name newservice.uat.example.com;

  ssl_certificate     /etc/letsencrypt/live/newservice.uat.example.com/fullchain.pem;
  ssl_certificate_key /etc/letsencrypt/live/newservice.uat.example.com/privkey.pem;
  ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
  ssl_ciphers         HIGH:!aNULL:!MD5;

  location / {
    proxy_pass        http://<container_name>;
```

```
    proxy_set_header   X-Forwarded-Proto \(scheme;
    proxy_set_header   Upgrade \)http_upgrade;
    proxy_set_header   Connection keep-alive;
    proxy_set_header   Host \(host;
    proxy_cache_bypass \)http_upgrade;
    proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header   X-Forwarded-Proto https;
    proxy_set_header   X-Forwarded-Ssl on;
  }
}
```

Notes:

1. Replace <container_name> with the Docker container name running your service.
2. If the service exposes a specific internal port, append it in the proxy_pass URL, e.g., http://<container_name>:<container_port>.
3. Save and close the file.

3. Generate SSL Certificate

1. Install Certbot if it is not already installed:

```
sudo apt update
sudo apt install certbot python3-certbot-nginx -y
```

1. Generate the SSL certificate and configure Nginx automatically:

```
sudo certbot --nginx -d newservice.uat.example.com
```

1. Follow the interactive prompts to complete HTTPS configuration.

Tip: Enable automatic renewal of SSL certificates:

```
sudo crontab -e
# Add the following line to renew certificates daily
0 3 * certbot renew --quiet
```

4. Update Docker Compose

1. Stop current Docker services in UAT:

```
docker-compose -f docker-compose.yaml -f docker-compose.uat.yaml down
```

1. Bring up the updated services:

```
docker-compose -f docker-compose.yaml -f docker-compose.uat.yaml up -d
```

1. Verify that your new service container is running:

```
docker ps
```

5. Verify Deployment

1. Access the new service via the domain:

```
https://newservice.uat.example.com
```

1. Check logs if any issues occur:

```
docker logs <container_name>
sudo journalctl -u nginx -f
```

Best Practices

- Always back up nginx.uat.conf before making edits.
- Use docker-compose down only when necessary; up -d can often apply changes without downtime.
- Ensure firewall rules allow HTTP (80) and HTTPS (443) traffic.
- Monitor logs after deployment to catch configuration issues early.
- Schedule automatic SSL renewal using Certbot's cron job.

Article 03 / 26 Jun 2025 / .NET, Infrastructure / 2 min read

Handling forwarded headers in ASP.NET Core behind Nginx

Why your app sees the wrong scheme and client IP behind a proxy, and the few lines that fix it.

When hosting an ASP.NET Core application behind a reverse proxy like **Nginx**, you must configure your app to handle **forwarded headers** properly. This ensures that properties like `Request.Scheme`, `Request.Host`, and `HttpContext.Connection.RemoteIpAddress` reflect the **original client values**, not the proxy.

Why Forwarded Headers Matter

When a client makes a request to your app through a proxy (like Nginx), the original request metadata is hidden from the app unless forwarded explicitly. To preserve these details:

- **Nginx** forwards the headers (e.g., `X-Forwarded-For`, `X-Forwarded-Proto`, `X-Forwarded-Host`).
- **ASP.NET Core** must be told to trust and read those headers.

If you don't configure this correctly:

- `Request.Scheme` might return `http` even when the client used HTTPS.
- `Request.Host` may show `localhost` or proxy IP instead of the original host.
- `HttpContext.Connection.RemoteIpAddress` will reflect the proxy's IP.

Nginx Configuration for Forwarded Headers

Here is the correct Nginx `server` block configuration to forward the necessary headers:

```
server {
    listen 443 ssl;
    server_name your-domain.com;

    location / {
        proxy_pass http://localhost:5000;  # Your .NET app port
        proxy_http_version 1.1;

        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Forwarded-Host $host;
        proxy_cache_bypass $http_upgrade;
    }

    ssl_certificate /etc/letsencrypt/live/your-domain.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/your-domain.com/privkey.pem;
    include /etc/letsencrypt/options-ssl-nginx.conf;
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem;
}
```

> Remember to replace `your-domain.com` and app port appropriately.

ASP.NET Core Configuration

In your `Program.cs` or `Startup.cs`, you must enable forwarded headers middleware early in the pipeline:

```
app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor |
                       ForwardedHeaders.XForwardedProto |
                       ForwardedHeaders.XForwardedHost,
    KnownProxies = { IPAddress.Parse("127.0.0.1") } // or the IP of your
reverse proxy
});
```

Middleware Placement

Ensure `UseForwardedHeaders()` is placed **before** any middleware that relies on `HttpContext` (like `UseRouting`, `UseAuthentication`, etc.).

Security Consideration: Header Spoofing

ASP.NET Core **does not trust forwarded headers by default** for security reasons. Without validation, a malicious client could send spoofed `X-Forwarded-For` or `X-Forwarded-Proto` headers directly and trick your app.

How to Secure It

- Use `KnownProxies` if your proxy has a **fixed IP**:

```
options.KnownProxies.Add(IPAddress.Parse("127.0.0.1"));
```

- Use `KnownNetworks` if you're behind a load balancer or container network:

```
options.KnownNetworks.Add(new IPNetwork(IPAddress.Parse("10.0.0.0"),
8));
```

This ensures ASP.NET Core will only process forwarded headers from trusted sources.

Summary

- Configure **Nginx** to forward all necessary headers.
- Configure **ASP.NET Core** to read and trust those headers using `UseForwardedHeaders()`.
- Protect your app from **header spoofing** by explicitly listing `KnownProxies` or `KnownNetworks`.
- Always test your setup with:

```
sudo nginx -t && sudo systemctl reload nginx
```

With this setup, your ASP.NET Core app will correctly reflect the original request information, even when running behind a reverse proxy like Nginx.

Article 04 / 12 Jun 2025 / .NET, Infrastructure / 3 min read

Defending a .NET API against DDoS attacks

Layered protection from Nginx down to the app: limits, timeouts and what to log when traffic spikes.

A **DDoS attack** (Distributed Denial of Service) is a **malicious attempt to disrupt the normal traffic** of a targeted server, service, or network by **overwhelming it with a flood of internet traffic**.

🔍 How It Works:

- **Distributed**: The attack comes from **many different sources**, often from a network of infected computers (called a **botnet**).
- **Denial of Service**: The target becomes **unavailable to legitimate users** because it's overloaded with fake traffic.

💥 Common Types of DDoS Attacks:

1. **Volume-Based Attacks**
   - Goal: Consume all available bandwidth.
   - Example: UDP floods, ICMP floods.
   - Measured in: **Gbps (Gigabits per second)**
2. **Protocol Attacks**
   - Goal: Exploit weaknesses in Layer 3 and 4 protocols (e.g., TCP/IP).
   - Example: SYN flood, Ping of Death.
   - Measured in: **Packets per second (PPS)**
3. **Application Layer Attacks**
   - Goal: Target Layer 7 (HTTP/HTTPS) where web pages are generated.

- Example: HTTP flood, Slowloris.
- Measured in: **Requests per second (RPS)**

⚠️ Real-World Consequences:

- Downtime for websites or apps.
- Loss of revenue and reputation.
- Security teams may miss other intrusions while dealing with the flood.

🛡️ Prevention & Mitigation:

- Use **Web Application Firewalls (WAF)**.
- Employ **rate limiting** and **traffic filtering**.
- Partner with **CDNs** or DDoS protection services like Cloudflare or AWS Shield.
- Monitor traffic for abnormal patterns.

Let me know if you want a diagram or code-level example related to .NET or networking security practices for handling DDoS threats.

✅ Diagram: What a DDoS Attack Looks Like

Here's a basic representation:

```
[Bot 1]         [Bot 2]         [Bot 3]         [Bot N]
   |               |               |               |
   +---------------+---------------+---------------+
                                ↓
                  [Your .NET API Server (Target)]
                        ↑         ↑
                  Legitimate users get denied
```

🧰 Step-by-Step Mitigation in .NET Web API

While you can't stop a full-scale DDoS (especially at the infrastructure level) from code alone, you **can reduce the impact** with smart coding and proper configurations.

🔐 1. Rate Limiting

Use a middleware like `AspNetCoreRateLimit`.

🛠️ Setup:

Install via NuGet:

```
dotnet add package AspNetCoreRateLimit
```

Configure in `Program.cs`:

```
builder.Services.AddMemoryCache();
builder.Services.Configure<IpRateLimitOptions>
(builder.Configuration.GetSection("IpRateLimiting"));
builder.Services.AddInMemoryRateLimiting();
builder.Services.AddSingleton<IRateLimitConfiguration,
RateLimitConfiguration>();

app.UseIpRateLimiting();
```

Add to `appsettings.json`:

```
"IpRateLimiting": {
  "EnableEndpointRateLimiting": true,
  "StackBlockedRequests": false,
  "RealIpHeader": "X-Real-IP",
  "ClientIdHeader": "X-ClientId",
  "HttpStatusCode": 429,
  "GeneralRules": [
    {
      "Endpoint": "*",
      "Period": "1s",
      "Limit": 5
    },
    {
      "Endpoint": "*",
      "Period": "1m",
      "Limit": 100
    }
  ]
}
```

🛡️ 2. Throttle Resource-Intensive Endpoints

Limit or defer heavy operations like report generation, file uploads, or PDF creation:

```
[HttpPost]
[Route("generate-heavy-report")]
public async Task<IActionResult> GenerateReport(CancellationToken token)
{
    if (!IsRequestAllowed(Request))
        return StatusCode(429, "Too Many Requests");

    var report = await _reportService.GenerateAsync(token);
    return File(report, "application/pdf");
}
```

🔍 3. Use Application Firewall / CDN

Offload protection to services **before requests reach your .NET server**:

- 🛡️ **Cloudflare**, **AWS Shield**, or **Azure Front Door**
- 🌐 **Reverse proxy (e.g., NGINX or Azure Application Gateway)**

🧪 4. Detect Patterns via Logs

Use structured logging (e.g., **Serilog + Seq**) to identify high-frequency IPs.

```
Log.Information("Request from IP: {IP}",
HttpContext.Connection.RemoteIpAddress);
```

Later, you can aggregate and analyze in a dashboard like Kibana, Grafana, or Seq.

🧰 5. Caching to Reduce Repeated Computation

Use `MemoryCache` or `ResponseCache` for GET endpoints:

```
[HttpGet]
[ResponseCache(Duration = 60)]
public IActionResult GetPublicData()
{
    var data = _service.GetCachedData();
    return Ok(data);
}
```

⚙️ 6. Enable Connection Limits (Infrastructure Level)

On IIS / Kestrel / nginx, configure:

- Max connections
- Max requests per second
- Timeouts

For **Kestrel**, in `appsettings.json`:

```
"Kestrel": {
  "Limits": {
    "MaxConcurrentConnections": 100,
    "MaxRequestBodySize": 10485760
  }
}
```

📦 Optional: Cloudflare DDoS Protection

If you're hosting the API behind a domain, you can enable **Cloudflare's DDoS protection** in their dashboard, which will:

- Filter malicious traffic.
- Challenge bots.

- Log/report events.

✅ Summary Checklist

Defense	Level	Tool
Rate Limiting	App	`AspNetCoreRateLimit`
Firewall Rules	Infra	IIS / Azure / Cloudflare
Caching	App	`ResponseCache`, `IMemoryCache`
Logging & Detection	App	Serilog + Seq
CDN + Edge Protection	Edge	Cloudflare / Azure Front Door
Alerting	Monitoring	Application Insights / Prometheus

Article 05 / 19 Nov 2024 / Process / 3 min read

Documenting a system you inherited

How to write down an existing system so the next engineer doesn't have to reverse-engineer it.

To properly document the **existing Kubernetes environment setup** and replicate it to a new server, you need a comprehensive approach. This includes gathering details about the infrastructure, Kubernetes cluster, services, configurations, and deployment pipelines. Here's a step-by-step plan to help you document the current setup thoroughly and accurately.

Step 1: Gather Cluster Information

Check Cluster Details

Run the following commands to understand the existing cluster configuration:

- **Cluster Info:**

```
kubectl cluster-info
```

- **Node Information:**

```
kubectl get nodes -o wide
```

Document the following:

- Kubernetes version
- Node count and types (e.g., worker, master)
- Node OS and specifications (CPU, memory)

- **Namespace Overview:**

```
kubectl get namespaces
```

Document the namespaces in use (e.g., `default`, `dev`, `prod`).

Check Storage and Networking

- **Storage Classes:**

```
kubectl get storageclass
```

Document the available storage classes (e.g., `standard`, `fast`, `gp2`).

- **Network Configuration:**

Check the CNI plugin in use (e.g., Calico, Flannel):

```
kubectl get pods -n kube-system
```

Look for networking components like `calico-node` or `flannel`.

Step 2: Document Namespace-Specific Details

For each namespace (e.g., `dev`, `prod`), gather the following details:

Deployment Configurations

- **List Deployments:**

```
kubectl get deployments -n <namespace>
```

- **Describe a Deployment:**

```
kubectl describe deployment <deployment-name> -n <namespace>
```

- **Export Deployment YAML:**

```
kubectl get deployment <deployment-name> -n <namespace> -o yaml >
deployment.yaml
```

Document the following:

- Deployment name and replicas
- Image versions (tagged versions)
- Resource requests and limits (CPU, memory)
- Environment variables

Service and Ingress Configurations

- **List Services:**

```
kubectl get svc -n <namespace>
```

- **Describe a Service:**

```
kubectl describe svc <service-name> -n <namespace>
```

- **List Ingress Resources:**

```
kubectl get ingress -n <namespace>
```

- **Describe an Ingress:**

```
kubectl describe ingress <ingress-name> -n <namespace>
```

Document the following:

- Service types (ClusterIP, NodePort, LoadBalancer)
- Ports and target ports
- Ingress rules (hostnames, paths, TLS settings)

Step 3: Document ConfigMaps and Secrets

ConfigMaps and Secrets store configuration data for your services.

- **List ConfigMaps:**

```
kubectl get configmap -n <namespace>
```

- **Export ConfigMap YAML:**

```
kubectl get configmap <configmap-name> -n <namespace> -o yaml >
configmap.yaml
```

- **List Secrets:**

```
kubectl get secret -n <namespace>
```

- **Export Secret YAML (base64 encoded):**

```
kubectl get secret <secret-name> -n <namespace> -o yaml > secret.yaml
```

Document:

- ConfigMap keys and values (non-sensitive information)
- Secrets usage (e.g., database passwords, API keys)

Note: Avoid storing Secrets directly in version control; use tools like Helm or Kubernetes Secret management solutions (e.g., Sealed Secrets, HashiCorp Vault).

Step 4: Document Persistent Volumes (PV) and Persistent Volume Claims (PVC)

- **List Persistent Volumes:**

```
kubectl get pv
```

- **List Persistent Volume Claims:**

```
kubectl get pvc -n <namespace>
```

Document:

- Volume types (e.g., `nfs`, `hostPath`, `gp2`)
- Bound claims and storage capacity
- Access modes (ReadWriteOnce, ReadOnlyMany)

Step 5: Document CI/CD Pipeline

If the deployment uses a CI/CD pipeline (e.g., GitHub Actions, Jenkins, GitLab CI):

- Check the pipeline configuration files (`.yml` files) in the repository.
- Document the stages (build, test, deploy).
- Identify triggers (manual, automatic based on branch/tag push).
- Note any environment variables or secrets used in the pipeline.

Example GitHub Actions configuration:

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2
      - name: Build Docker image
        run: docker build -t myapp:${{ github.sha }} .
      - name: Push to registry
        run: docker push myapp:${{ github.sha }}
      - name: Deploy to Kubernetes
        run: kubectl apply -f k8s/deployment.yaml
```

Step 6: Document Monitoring and Logging Setup

Check the monitoring and logging setup in the current environment:

- **Monitoring:** Identify tools like Prometheus, Grafana, or Datadog.
  - Check for existing Prometheus configuration:

```
kubectl get pods -n monitoring
```

- **Logging:** Identify tools like ELK stack (Elasticsearch, Logstash, Kibana), Loki, or Fluentd.
  - Check Fluentd or logging DaemonSet:

```
kubectl get daemonset -n logging
```

Document:

- Monitoring and logging stack components
- Alerts and dashboards configuration
- Data retention policies

Step 7: Document Rollback and Disaster Recovery Plan

- **Rollback Strategy:** Note how to revert to the previous version (using Helm rollback or previous deployment YAML files).
- **Backup Strategy:** Document how to back up persistent data (e.g., volume snapshots, database backups).
- **Disaster Recovery:** Document the steps for recovering the cluster or data in case of failure.

Example Rollback Command:

```
kubectl rollout undo deployment <deployment-name> -n <namespace>
```

Step 8: Export the Current Cluster State

For a complete backup of the current environment, export the entire namespace configuration:

```
kubectl get all -n <namespace> -o yaml > namespace-backup.yaml
```

You can also use **Helm** if the applications are deployed using Helm charts:

```
helm list -n <namespace>
helm get all <release-name> -n <namespace>
```

Step 9: Create Infrastructure as Code (IaC) Documentation

If possible, use tools like **Helm**, **Kustomize**, or **Terraform** to define the infrastructure setup as code. This makes the process repeatable and consistent across environments.

Example Helm Chart:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 3
  template:
    spec:
      containers:
        - name: myapp
          image: myapp:latest
```

Conclusion

By following these steps, you will have a comprehensive understanding of the current environment and can easily replicate it on a new server. Documenting your environment setup helps ensure consistency, reduces the risk of misconfigurations, and simplifies future migrations or scaling efforts.

Article 06 / 5 Nov 2024 / .NET, Infrastructure / 6 min read

Nginx and free SSL for a .NET app with Certbot

Nginx as a reverse proxy for .NET, first over HTTP, then with a Let's Encrypt certificate that renews itself.

In this article, we'll cover a comprehensive guide to setting up Nginx as a reverse proxy for a .NET application, running it over HTTP initially, and then adding SSL encryption with a free certificate from Let's Encrypt using Certbot. We'll also discuss automating certificate renewal to keep your site secure without manual intervention.

Prerequisites

Before we start, make sure you have:

1. A Linux server (Ubuntu or Debian is recommended).
2. A .NET application ready to be deployed.
3. Nginx installed on your server.
4. A registered domain name pointed to your server's IP address.

Why You Should Use Nginx for Your .NET Application and How to Set It Up with SSL

In today's fast-paced digital landscape, delivering a reliable, secure, and efficient web experience is essential. That's where **Nginx** comes in—a powerful web server and reverse proxy known for its high performance and flexibility. This article will discuss the key benefits of using Nginx for your .NET application, walk you through the setup process, and guide you on adding SSL encryption using Certbot to secure your site with HTTPS.

Why Use Nginx?

Nginx (pronounced "engine-x") has quickly become one of the most popular web servers worldwide, particularly favored for high-traffic websites and applications requiring exceptional performance. Here's why Nginx is a valuable addition to your application stack:

1. High Performance and Speed

- Nginx's event-driven architecture efficiently manages multiple concurrent connections, making it capable of handling large amounts of traffic without overloading the server. This results in faster response times, even under heavy load.

2. Reverse Proxy and Load Balancing

- Nginx acts as a reverse proxy, sitting between clients and backend servers (such as your .NET application). It distributes incoming traffic across multiple servers, improving load handling and reducing the risk of downtime.
- Load balancing with Nginx enhances both performance and reliability by routing requests to available servers, helping to maintain uptime even if one server fails.

3. Enhanced Security

- By serving as a barrier between users and backend servers, Nginx hides backend details, improving application security. Its ability to handle SSL/TLS makes it easy to secure traffic, protecting against eavesdropping and man-in-the-middle attacks.

4. Efficient Static Content Delivery

- Nginx is highly efficient at serving static files (e.g., images, CSS, JavaScript) from cache, reducing the load on backend servers and speeding up load times for end users.

5. Resource Efficiency

- Nginx's lightweight resource usage makes it ideal for environments with limited resources or high-performance requirements, helping optimize server costs and efficiency.

6. SEO and Caching Support

- Nginx enables server-level caching, improving load times and user experience. Faster websites also benefit from improved SEO rankings, as search engines prioritize sites with quick load times.

7. Easy SSL/TLS Integration

- With tools like Certbot, you can set up free SSL certificates from Let's Encrypt in minutes, making it easy to switch to HTTPS and meet modern security standards.

Here are the steps to follow for deploying Nginx:

Step 1: Install Nginx on Linux

Nginx will serve as the reverse proxy for your .NET application, handling requests and forwarding them to your application.

1. **Update your package index**:

```
sudo apt update
```

2. **Install Nginx**:

```
sudo apt install nginx -y
```

3. **Enable and Start Nginx**:

```
sudo systemctl enable nginx
sudo systemctl start nginx
```

4. **Verify the Installation**: Check if Nginx is running by visiting your server's IP address in a browser (`http://your_server_ip`). You should see the Nginx welcome page if everything is set up correctly.

Step 2: Configure Nginx for Your .NET Application

Now, let's configure Nginx to serve your .NET application over HTTP.

1. **Create a New Nginx Configuration File**:
   Open a new configuration file for your site:

```
sudo nano /etc/nginx/sites-available/myapp
```

2. **Add Configuration for the Reverse Proxy**:
   Replace the contents of the file with the following configuration, adjusting `server_name` to your domain and `proxy_pass` to the port your .NET app runs on (commonly 5000 for Kestrel):

```
server {
    listen 80;
    server_name example.com www.example.com;

    location / {
        proxy_pass http://localhost:5000;  # Use the port your .NET app
is running on
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

3. **Enable the Configuration**:
   Link this configuration file to the Nginx sites-enabled directory:

```
sudo ln -s /etc/nginx/sites-available/myapp /etc/nginx/sites-enabled/
```

4. **Test and Reload Nginx**:
   Test your configuration for syntax errors, then reload Nginx to apply the changes:

```
sudo nginx -t
sudo systemctl reload nginx
```

At this stage, your .NET application should be accessible via HTTP at `http://your_domain`.

Step 3: Install Certbot and Obtain an SSL Certificate

Now that your site is accessible over HTTP, let's secure it with SSL using Certbot and Let's Encrypt.

1. **Install Certbot**:
   Update your package list and install Certbot with the Nginx plugin:

```
sudo apt update
sudo apt install certbot python3-certbot-nginx -y
```

2. **Obtain an SSL Certificate**:
   Certbot will help automate the SSL configuration for Nginx.

```
sudo certbot --nginx
```

   - Follow the prompts to enter your email address and agree to the terms of service.
   - Certbot will detect your domain from the Nginx configuration and ask if you want to redirect all HTTP traffic to HTTPS. Choose to redirect, as this ensures all traffic is encrypted.

3. **Verify SSL Configuration**:
   Once complete, Certbot will update your Nginx configuration to serve your .NET application securely over HTTPS. You can test this by visiting:

```
https://your_domain
```

   You should see a padlock icon in the browser, confirming the SSL setup.

Step 4: Update Nginx Configuration (if needed)

If you want to customize your Nginx configuration after installing SSL, you might want to check or update the SSL settings in your Nginx configuration file located in `/etc/nginx/sites-available/myapp`:

```
server {
    listen 80;
    server_name example.com www.example.com;

    # Redirect HTTP to HTTPS
    return 301 https://$host$request_uri;
}

server {
    listen 443 ssl;
    http2 on;
    server_name example.com www.example.com;

    ssl_certificate /etc/letsencrypt/live/example.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/example.com/privkey.pem;

    location / {
        proxy_pass http://localhost:5000; # Change this if your app
runs on a different port
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

Step 5: Test and Reload Nginx

After updating your configuration, always test it for syntax errors and then reload Nginx:

```
sudo nginx -t
sudo systemctl reload nginx
```

Step 6: Automatic Renewal of SSL Certificates

1. Let's Encrypt certificates are valid for 90 days, but Certbot can automate the renewal process.

1. **Test Automatic Renewal**:
   You can simulate a renewal test with the following command:

```
sudo certbot renew --dry-run
```

   If you see no errors, automatic renewal is working correctly.

2. **Check Cron Jobs (Optional)**:
   Certbot is typically set up to run as a cron job automatically. You can check for a cron job by running:

```
sudo crontab -l
```

   If you don't see an entry for Certbot, you can add one manually. For example, to run the renewal command every day at noon:

```
0 12 * * * certbot renew --quiet
```

Step 1: Open the Crontab File

You need to edit the crontab file for the root user since Certbot requires administrative privileges to run the renewal process. Use the following command:

```
sudo crontab -e
```

Step 2: Add the Cron Job

1. This will open the crontab file in a text editor (usually `nano` by default).
2. Scroll to the bottom of the file and add the following line:

```
0 12 * * * certbot renew --quiet
```

- This line schedules the renewal command to run every day at 12:00 PM (noon).
- The `--quiet` option prevents output unless there are errors.

Step 3: Save and Exit

- If you're using `nano`, save the changes by pressing `CTRL + X`, then `Y`, and then `Enter`.
- If you're using another editor, follow the respective steps to save and exit.

Step 4: Verify the Cron Job

You can verify that the cron job has been added by listing the cron jobs for the root user:

```
sudo crontab -l
```

This command will display all scheduled cron jobs for the root user, and you should see your new entry.

Additional Notes

- **Log Output (Optional)**: If you want to log the output of the cron job (for troubleshooting purposes), you can modify the cron job like this:

  ```
  0 12 * * * certbot renew --quiet >> /var/log/certbot-renew.log 2>&1
  ```

  This will append the output of the command to a log file located at `/var/log/certbot-renew.log`.
- **Systemd Timer (Alternative)**: If your system uses `systemd`, you might already have a timer set up for Certbot. You can check if it's active by running:

  ```
  systemctl list-timers
  ```

  Look for `certbot.timer` in the output. This is an alternative to using cron jobs for managing scheduled tasks.

Conclusion

Congratulations! You've successfully set up Nginx to serve your .NET application, secured it with a Let's Encrypt SSL certificate, and automated the renewal process with Certbot. By following this setup, you ensure that your .NET application remains secure and accessible over HTTPS.

If you have further questions or encounter issues, feel free to ask for more guidance. Happy hosting!

Article 07 / 21 Aug 2024 / .NET / 3 min read

Mastering HttpClient in .NET with Refit

Socket exhaustion, IHttpClientFactory and typed clients with Refit: the mistakes everyone makes once.

Introduction

In the .NET ecosystem, `HttpClient` is the go-to tool for making HTTP requests to web services. However, despite its simplicity, improper usage can lead to significant issues such as socket exhaustion and poor performance. In this article, we will explore the different ways to implement `HttpClient`, discuss best practices, and introduce Refit—a powerful library that simplifies API consumption in .NET.

Why HttpClient is Important

`HttpClient` is a versatile class that provides a straightforward way to send HTTP requests and receive responses from a resource identified by a URI. It's essential in any application that interacts with RESTful APIs, third-party services, or even microservices within a distributed system.

However, as a senior software engineer, it's crucial to understand the implications of how `HttpClient` is instantiated and managed within an application. Mismanagement can lead to serious performance bottlenecks.

Common Pitfalls with HttpClient

1. Socket Exhaustion

- **Problem**: Creating a new `HttpClient` instance for each request can exhaust the number of available sockets, leading to `SocketException`.
- **Cause**: Each `HttpClient` instance consumes a socket, which isn't immediately released upon disposal, especially under high load.

- **Solution**: Use a singleton `HttpClient` or leverage `IHttpClientFactory`.

2. DNS Caching

- **Problem**: The default `HttpClient` implementation caches DNS entries indefinitely, which can cause requests to fail if the underlying IP changes.
- **Solution**: `IHttpClientFactory` handles DNS updates by pooling `HttpClientHandler` instances.

Different Ways to Implement HttpClient

1. Basic Instantiation

This is the most straightforward approach but should be used cautiously.

```
using (var client = new HttpClient())
{
    var response = await client.GetAsync("https://api.example.com/data");
    response.EnsureSuccessStatusCode();
    var content = await response.Content.ReadAsStringAsync();
}
```

Pros:

- Simple and easy to understand.

Cons:

- Risk of socket exhaustion if used in a high-traffic environment.

2. Singleton HttpClient

To avoid socket exhaustion, use a singleton `HttpClient` instance.

```
public class MyService
{
    private static readonly HttpClient _httpClient = new HttpClient();

    public async Task<string> GetDataAsync()
    {
        var response = await
_httpClient.GetAsync("https://api.example.com/data");
        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}
```

Pros:

- Prevents socket exhaustion.

Cons:

- Potential DNS caching issues.

3. Using IHttpClientFactory (Recommended)

Introduced in .NET Core 2.1, `IHttpClientFactory` addresses the issues of socket exhaustion and DNS caching by managing `HttpClient` instances efficiently.

```
public class MyService
{
    private readonly HttpClient _httpClient;

    public MyService(IHttpClientFactory httpClientFactory)
    {
        _httpClient = httpClientFactory.CreateClient("MyClient");
    }

    public async Task<string> GetDataAsync()
    {
        var response = await
_httpClient.GetAsync("https://api.example.com/data");
        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}
```

Pros:

- Handles DNS updates automatically.
- Prevents socket exhaustion.
- Supports named and typed clients for better configuration.

Cons:

- Slightly more complex to set up compared to a basic instantiation.

4. Typed Clients

Typed clients are a way to create strongly-typed wrappers around `HttpClient` using `IHttpClientFactory`.

```
public class MyApiClient
{
    private readonly HttpClient _httpClient;

    public MyApiClient(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }

    public async Task<string> GetDataAsync()
    {
        var response = await
_httpClient.GetAsync("https://api.example.com/data");
        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}

// Registration in Startup.cs
services.AddHttpClient<MyApiClient>();
```

Pros:

- Clean, reusable, and testable code.
- Centralized configuration.

Cons:

- Requires additional setup in the DI container.

Advanced API Consumption with Refit

Refit is a type-safe REST client for .NET that simplifies making HTTP requests by automatically generating API client implementations from an interface. It abstracts much of the boilerplate code associated with `HttpClient`.

1. Getting Started with Refit

To use Refit, first, install the NuGet package:

```
dotnet add package Refit
```

2. Defining an API Interface

```
public interface IMyApi
{
    [Get("/data")]
    Task<string> GetDataAsync();
}
```

3. Consuming the API

```
public class MyService
{
    private readonly IMyApi _myApi;

    public MyService(IMyApi myApi)
    {
        _myApi = myApi;
    }

    public async Task<string> GetDataAsync()
    {
        return await _myApi.GetDataAsync();
    }
}

// Registration in Startup.cs
services.AddRefitClient<IMyApi>()
        .ConfigureHttpClient(c => c.BaseAddress = new
Uri("https://api.example.com"));
```

Pros:

- Clean and concise API consumption.
- Strongly typed, making it easier to refactor.
- Built-in support for error handling, JSON serialization, and retries.

Cons:

- Additional abstraction layer; less control over `HttpClient` configuration.

Best Practices

1. **Prefer IHttpClientFactory**: Always prefer `IHttpClientFactory` for managing `HttpClient` instances to avoid common pitfalls.
2. **Use Typed Clients**: For complex or reusable `HttpClient` logic, prefer typed clients for better maintainability.
3. **Leverage Refit for Simplicity**: If you're working with multiple APIs and want to minimize boilerplate, Refit is an excellent choice.
4. **Handle Timeouts and Retries**: Always configure timeouts and consider implementing retry policies for transient errors.
5. **Dispose HttpClient Correctly**: Avoid using `using` statements with `HttpClient` unless it's necessary for a short-lived operation. Otherwise, prefer long-lived instances.
6. **Secure Sensitive Data**: Use secure mechanisms for handling sensitive data such as API keys or tokens within your `HttpClient` requests.

Conclusion

`HttpClient` is a powerful tool in the .NET developer's arsenal, but with great power comes great responsibility. By following best practices such as using `IHttpClientFactory` and typed clients, you can avoid common pitfalls like socket exhaustion and DNS issues. Additionally, Refit offers a streamlined approach to API consumption, making your code cleaner and more maintainable.

By mastering these techniques, you'll ensure that your .NET applications are robust, efficient, and scalable, meeting the demands of modern software development.

This article provides a comprehensive guide for .NET developers, from basic `HttpClient` usage to advanced techniques with Refit, ensuring that your applications are both performant and maintainable.

Article 08 / 20 Aug 2024 / Databases / 2 min read

Triggers in Postgres

Row and statement level triggers, and when an invariant belongs in the database rather than the app.

The trigger is a function that invokes automatically when an event associated with a table occurs like an update, delete, or insert.

Triggers can be defined to execute either at the statement level or the row level.

1. Statement-level triggers: Statement-level triggers are executed once for each triggering SQL statement, regardless of the number of rows affected by the statement. Statement-level triggers are useful when the action to be taken by the trigger is independent of the number of rows being affected. Statement-level triggers are defined using the `FOR EACH STATEMENT` clause.
2. Row-level triggers: Row-level triggers are executed once for each row affected by the triggering SQL statement. Row-level triggers are useful when the action to be taken by the trigger depends on the data in the affected rows. Row-level triggers are defined using the `FOR EACH ROW` clause.

Types of Trigger

1. Before Trigger Example: Suppose we have a table called "employees" with columns "id", "name", and "salary", and we want to make sure that the salary of any employee being updated is not less than 1000. We can create a before trigger to perform this validation:

```
CREATE OR REPLACE FUNCTION validate_salary()
RETURNS trigger AS
$BODY$
BEGIN
    IF NEW.salary < 1000 THEN
        RAISE EXCEPTION 'Salary cannot be less than 1000';
    END IF;
    RETURN NEW;
END;
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER check_salary
BEFORE UPDATE ON employees
FOR EACH ROW
EXECUTE FUNCTION validate_salary();
```

This trigger function will be executed before any update on the "employees" table, and will check if the new salary value is less than 1000. If so, it will raise an exception and prevent the update from occurring.

1. After Trigger Example: Suppose we have a table called "orders" with columns "id", "customer_id", "total", and "date", and we want to log any changes made to this table. We can create an after trigger to perform this logging:

```
CREATE OR REPLACE FUNCTION log_changes()
RETURNS trigger AS
$BODY$
BEGIN
    IF TG_OP = 'INSERT' THEN
        INSERT INTO order_logs (order_id, event, date)
        VALUES (NEW.id, 'inserted', now());
    ELSIF TG_OP = 'UPDATE' THEN
        INSERT INTO order_logs (order_id, event, date)
        VALUES (NEW.id, 'updated', now());
    ELSIF TG_OP = 'DELETE' THEN
        INSERT INTO order_logs (order_id, event, date)
        VALUES (OLD.id, 'deleted', now());
    END IF;
    RETURN NEW;
END;
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER orders_log
AFTER INSERT OR UPDATE OR DELETE ON orders
FOR EACH ROW
EXECUTE FUNCTION log_changes();
```

This trigger function will be executed after any insert, update, or delete on the "orders" table, and will log the event (inserted, updated, or deleted) and the current date/time into an "order_logs" table.

1. Instead of Trigger Example: Suppose we have a view called "customers_orders" that combines the "customers" and "orders" tables to show the customer name, order total, and order date. We want to make sure that any updates to this view only affect the "orders" table, not the "customers" table. We can create an instead of trigger to handle this:

```
CREATE OR REPLACE FUNCTION update_orders()
RETURNS trigger AS
$BODY$
BEGIN
    UPDATE orders
    SET total = NEW.total, date = NEW.date
    WHERE id = NEW.id;
    RETURN NULL;
END;
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER customers_orders_update
INSTEAD OF UPDATE ON customers_orders
FOR EACH ROW
EXECUTE FUNCTION update_orders();
```

This trigger function will be executed instead of any update on the "customers_orders" view, and will update the "orders" table based on the new values provided. It will return NULL to indicate that the view should not be updated directly. This way, any updates to the "customers_orders" view will only affect the "orders" table, not the "customers" table.

percona

Article 09 / 19 Aug 2024 / .NET, Infrastructure / 4 min read

Exploring YARP, the reverse proxy for .NET

Routing, load balancing and a gateway you configure in C#, for teams that already live in .NET.

Introduction

In modern web development, as applications scale, the need for efficient routing, load balancing, and reverse proxying becomes crucial. A reverse proxy can be a game-changer, helping to manage traffic, improve performance, and add an extra layer of security to your applications. Enter YARP (Yet Another Reverse Proxy), a powerful and flexible reverse proxy library developed by Microsoft specifically for .NET developers. This article explores what YARP is, why you should consider using it, and how to get started with it in your .NET projects.

What is YARP?

YARP is a highly customizable reverse proxy toolkit for building fast and reliable proxies in .NET. Built on top of ASP.NET Core, it leverages the underlying framework's performance and scalability while providing a flexible way to define routing rules and load balancing strategies.

One of the standout features of YARP is its extensibility. Unlike traditional reverse proxies, YARP is designed to be modified and extended by developers. You can customize its behavior using .NET code, making it a perfect fit for scenarios where out-of-the-box solutions might fall short.

Why Use a Reverse Proxy?

Before diving into YARP, it’s essential to understand why you might want to use a reverse proxy in the first place. A reverse proxy sits between clients and your backend servers, forwarding client requests to the appropriate server. This setup offers several benefits:

1. **Load Balancing**: Distribute incoming traffic across multiple backend servers, ensuring no single server is overwhelmed, which leads to better performance and reliability.
2. **Security**: By routing all traffic through a reverse proxy, you can hide the identity and internal structure of your backend servers. This adds an additional security layer, as clients interact only with the proxy.

3. **Caching**: Reverse proxies can cache responses from your backend servers, reducing the load on those servers and improving response times for clients.
4. **Routing**: You can direct traffic to different backend services based on URL patterns, headers, or other criteria, allowing you to manage multiple services or microservices with a single proxy.
5. **SSL Termination**: Reverse proxies can handle SSL termination, decrypting incoming requests before forwarding them to backend services, reducing the computational load on your servers.

Getting Started with YARP

Now that we understand the benefits of using a reverse proxy, let's see how you can implement YARP in your .NET applications.

Installation

To start using YARP in your project, you need to add the YARP NuGet package to your .NET project. You can do this via the .NET CLI or by adding the package directly through your project's package manager.

```
dotnet add package Microsoft.ReverseProxy
```

Basic Configuration

YARP is configured primarily through your `appsettings.json` file, where you define routes and clusters. A route in YARP specifies how requests should be matched and routed to a backend cluster. A cluster represents one or more destinations (servers) where the traffic will be forwarded.

Here's a basic example of a YARP configuration in `appsettings.json`:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ReverseProxy": {
    "Routes": {
      "users-route": {
        "ClusterId": "users-cluster",
        "Match": {
          "Path": "users-api/{**catch-all}"
        },
        "Transforms": [
          {
            "PathPattern": "{**catch-all}"
          }
        ]
      }
    },
    "Clusters": {
      "users-cluster": {
        "Destinations": {
          "destination1": {
            "Address": "https://localhost:7247/"
          }
        }
      }
    }
  }
}
```

Understanding the Configuration

Let's break down the configuration:

- **Routes**: Defines how incoming requests should be routed. In the example above, we have a route named `users-route` that matches any request path starting with `users-api/`. The

`ClusterId` property links this route to a cluster (`users-cluster`), which defines the destination servers.

- **Clusters**: Specifies one or more destinations (servers) that should receive the requests. The `users-cluster` cluster in the example has a single destination with an address of `https://localhost:7247/` .
- **Transforms**: YARP supports request transformations, allowing you to modify the request before forwarding it to the backend. In this example, the transform ensures that the matched path is passed along to the destination.

Adding YARP to the Middleware Pipeline

After configuring YARP in `appsettings.json` , you need to register it in your `Startup.cs` (or `Program.cs` in .NET 6 and later) file to make it part of the middleware pipeline.

Here's how you can do that:

```
namespace Gateway.Api;

public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        // Add services to the container.

        builder.Services.AddControllers();
        // Learn more about configuring Swagger/OpenAPI at
https://aka.ms/aspnetcore/swashbuckle
        builder.Services.AddEndpointsApiExplorer();
        builder.Services.AddSwaggerGen();

        builder.Services.AddReverseProxy()

.LoadFromConfig(builder.Configuration.GetSection("ReverseProxy"));

        var app = builder.Build();

        // Configure the HTTP request pipeline.
        if (app.Environment.IsDevelopment())
        {
            app.UseSwagger();
            app.UseSwaggerUI();
        }

        app.UseHttpsRedirection();

        app.MapReverseProxy();

        app.MapControllers();

        app.Run();
    }
}
```

In this setup:

- `services.AddReverseProxy()` registers the YARP services with the dependency injection container.
- `LoadFromConfig` tells YARP to load the configuration from the `appsettings.json` file.
- `app.MapReverseProxy()` enable YARP in the middleware pipeline, allowing it to handle incoming requests.

Advanced Scenarios with YARP

While the basic setup will cover many use cases, YARP's true power lies in its extensibility and advanced features.

- **Custom Load Balancing**: YARP supports custom load-balancing strategies. You can implement your own strategy by extending YARP's default behavior.
- **Health Checks**: YARP can monitor the health of your destinations and avoid routing traffic to unhealthy servers.
- **Rate Limiting**: You can apply rate limiting to control the flow of traffic to your backend services, protecting them from being overwhelmed.
- **Custom Middleware**: Since YARP is built on ASP.NET Core, you can leverage custom middleware to add logging, authentication, or any other functionality before or after requests are handled by YARP.

Conclusion

YARP is a robust and flexible reverse proxy solution tailored for .NET developers. Whether you need simple routing, complex load balancing, or advanced transformations, YARP provides the tools you need to build a performant and scalable reverse proxy. By leveraging YARP, you can ensure your applications are ready to handle the demands of modern web traffic, all while staying within the familiar .NET ecosystem.

Whether you're working on a monolithic application that needs to scale or managing a microservices architecture, YARP offers a compelling solution for reverse proxying that integrates seamlessly with your .NET projects.

So, if you haven't explored YARP yet, now is the time to dive in and see how it can help simplify and enhance your application's architecture. Happy coding!

Article 10 / 15 Aug 2024 / .NET / 2 min read

Cancellation tokens: a practical guide

Stop wasted work when a request is abandoned, from the controller down to the database call.

Cancellation Token in .NET: A Practical Guide

Cancellation tokens are a powerful feature in .NET that allow you to manage and gracefully cancel long-running tasks. Whether you're dealing with API requests, database operations, or any asynchronous processing, cancellation tokens help you maintain responsiveness and optimize resource usage. In this blog, we'll explore how cancellation tokens work and provide a practical example of their usage in a .NET API.

Table of Contents

1. **Understanding Cancellation Tokens**
2. **How Cancellation Tokens Work with Frontend/Browser**
3. **Implementing Cancellation Tokens in .NET**
4. **A Real-World Example**
5. **Conclusion**

1. Understanding Cancellation Tokens

Cancellation tokens in .NET are primarily used to signal and manage the cancellation of tasks. They are particularly useful in scenarios where a task might no longer be needed or where it's essential to free up resources if a client disconnects or a timeout occurs.

Key components:

- `CancellationTokenSource`: This creates a cancellation token and manages the cancellation.
- `CancellationToken`: This is the token itself, which is passed to tasks that might need to be canceled.

- `OperationCanceledException` : This exception is thrown when a task is canceled using a cancellation token.

2. How Cancellation Tokens Work with Frontend/Browser

Although cancellation tokens are a server-side feature, they work closely with the frontend in specific scenarios:

- **Client Disconnection**: If a user closes a browser tab or navigates away, the server can detect this and cancel ongoing tasks using the `HttpContext.RequestAborted` cancellation token.
- **Timeouts**: When a frontend sets a timeout for an HTTP request, the server detects this cancellation through the same token mechanism.
- **User-Initiated Cancellation**: In advanced scenarios, the frontend might provide the option to cancel an operation, which the backend can handle using a cancellation token.

3. Implementing Cancellation Tokens in .NET

To implement cancellation tokens in your .NET API, follow these steps:

1. **Create and Pass the Cancellation Token**
   Ensure that every async method accepts a `CancellationToken` parameter to allow for potential cancellation.

```
 public async Task<IActionResult> GetDataAsync(CancellationToken
cancellationToken)
 {
     var data = await _dataService.GetDataAsync(cancellationToken);
     return Ok(data);
 }
```

2. **Handle Cancellation Gracefully**
   Within your tasks, periodically check the cancellation token to exit early if a cancellation has been requested.

```
 public async Task<List<MyData>> GetDataAsync(CancellationToken
cancellationToken)
 {
     await Task.Delay(5000, cancellationToken);
     cancellationToken.ThrowIfCancellationRequested();
     return await _context.MyData.ToListAsync(cancellationToken);
 }
```

3. **Pass Cancellation Token in HTTP Client Calls**

   When making HTTP requests using `HttpClient`, pass the cancellation token to handle request cancellations properly.

```
 public async Task<HttpResponseMessage>
GetExternalDataAsync(CancellationToken cancellationToken)
 {
     var request = new HttpRequestMessage(HttpMethod.Get,
"https://api.example.com/data");
     return await _httpClient.SendAsync(request, cancellationToken);
 }
```

4. A Real-World Example

Let's dive into a practical example where a cancellation token is used in a .NET API to handle a request that fetches data from a database.

```
[ApiController]
[Route("api/[controller]")]
public class DataController : ControllerBase
{
    private readonly IDataService _dataService;

    public DataController(IDataService dataService)
    {
        _dataService = dataService;
    }

    [HttpGet("data")]
    public async Task<IActionResult> GetData(CancellationToken
cancellationToken)
    {
        try
        {
            // Fetch data with cancellation support
            var data = await _dataService.GetDataAsync(cancellationToken);
            return Ok(data);
        }
        catch (OperationCanceledException)
        {
            // Handle the cancellation request
            return StatusCode(StatusCodes.Status499ClientClosedRequest,
"Request was canceled.");
        }
    }
}

public class DataService : IDataService
{
    private readonly ApplicationDbContext _context;

    public DataService(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<List<MyData>> GetDataAsync(CancellationToken
cancellationToken)
```

```
    {
        // Simulate a delay and then fetch data from the database
        await Task.Delay(3000, cancellationToken);
        return await _context.MyData.ToListAsync(cancellationToken);
    }
}
```

5. Conclusion

Cancellation tokens are an essential feature in .NET for handling the cancellation of tasks gracefully, especially in API-driven applications. By understanding and implementing them properly, you can make your applications more responsive, resource-efficient, and capable of handling real-world scenarios like client disconnections and timeouts.

In this blog, we've covered the basics of cancellation tokens, how they interact with frontend requests, and a practical example of their use in a .NET API. Implementing cancellation tokens can significantly improve the performance and reliability of your applications, ensuring that resources are used efficiently and that your services can handle interruptions gracefully.

With this guide, you should now have a good understanding of how to use cancellation tokens in your .NET applications. Whether you're developing APIs or working with long-running tasks, cancellation tokens are a valuable tool to keep in your toolkit.

Article 11 / 15 Aug 2024 / .NET / 8 min read

Rate limiting in .NET, from concept to code

Fixed and sliding windows, token buckets and concurrency limits, with the built-in ASP.NET Core middleware.

In today's world of API-driven architectures, ensuring the stability, security, and availability of services is paramount. One essential tool for achieving this is **rate limiting**. Rate limiting helps you control the amount of incoming traffic to your API, protecting your application from abuse, preventing service overload, and ensuring fair usage among users.

In this blog post, we'll explore the concepts behind rate limiting, discuss different techniques to achieve it, and provide a detailed implementation of chained rate limiting using the Fixed Window technique and rate limiting by IP, user, or API key in .NET.

Table of Contents

1. **What is Rate Limiting?**
2. **Why is Rate Limiting Important?**
3. **Rate Limiting Techniques**
   - Fixed Window Rate Limiting
   - Sliding Window Rate Limiting
   - Token Bucket Rate Limiting
   - Leaky Bucket Rate Limiting
   - Dynamic Rate Limiting
   - Rate Limiting by IP, User, or API Key
4. **Implementing Rate Limiting in .NET**
   - Chained Rate Limiting with Fixed Window
   - Rate Limiting by IP, User, or API Key
5. **AutoReplenishment in Rate Limiting**
6. **Conclusion**

1. What is Rate Limiting?

Rate limiting is the practice of controlling the number of requests an API can handle within a specific time period. It acts as a gatekeeper, allowing only a certain number of requests from a client within a given timeframe. This helps protect your service from being overwhelmed by too many requests at once, ensuring that resources are allocated fairly and your application remains available and responsive.

2. Why is Rate Limiting Important?

- **Prevents Abuse**: Protects your API from being overwhelmed by malicious users or bots sending an excessive number of requests.
- **Ensures Fair Usage**: Helps enforce fair usage policies, ensuring that no single user or client consumes more than their share of resources.
- **Enhances Stability**: Prevents service degradation by controlling the flow of requests and avoiding server overload.
- **Security**: Deters denial-of-service (DoS) attacks and other forms of cyber-attacks that aim to exhaust your system's resources.

3. Rate Limiting Techniques

Rate limiting is a strategy used to control the rate at which a service or API can be accessed. This helps prevent abuse, ensure fair usage, and protect resources. There are various rate limiting techniques, each suited to different scenarios. Here's an overview of the most common rate limiting techniques:

1. Fixed Window Rate Limiting

- **Concept**: Requests are counted within a fixed time window (e.g., 1 minute, 1 hour). Once the request count exceeds the limit within that window, additional requests are blocked until the window resets.
- **Example**: Allow 100 requests per minute. After 100 requests, further requests are blocked until the next minute.

- **Use Case**: Simple and easy to implement but can cause bursts at the edges of the time window (e.g., at the end and start of consecutive windows).

2. Sliding Window Log Rate Limiting

- **Concept**: Instead of fixed intervals, the sliding window keeps a log of request timestamps within a rolling time window. Requests are counted based on the number of timestamps within the current window.
- **Example**: Allow 100 requests per 60 seconds, but the count is based on a sliding window of the last 60 seconds from the current time.
- **Use Case**: More precise than fixed window, prevents bursts, but can be memory intensive as it needs to store timestamps.

3. Sliding Window Counter Rate Limiting

- **Concept**: A more efficient version of the sliding window log, using two counters to track the number of requests in the current and previous windows, and a proportion based on the elapsed time.
- **Example**: Allow 100 requests per minute. If 30 seconds have passed, the count might allow a fraction of the 100 requests based on the time elapsed.
- **Use Case**: Balances accuracy and efficiency commonly used when memory is a concern but sliding behavior is desired.

4. Leaky Bucket Rate Limiting

- **Concept**: Requests are processed at a steady rate. The bucket represents the queue of requests, which “leaks” at a fixed rate. If the bucket overflows (i.e., too many requests), the excess requests are discarded or delayed.
- **Example**: Process 10 requests per second. Excess requests are either dropped or queued to be processed later.
- **Use Case**: Smooths out bursts, ensuring consistent traffic flow, suitable for scenarios where predictable traffic shaping is needed.

5. Token Bucket Rate Limiting

- **Concept**: A bucket is filled with tokens at a fixed rate. Each request consumes a token. If the bucket is empty, requests are either queued or rejected. The bucket can also accommodate burst requests if enough tokens are available.
- **Example**: 100 tokens are added to the bucket every minute. If the bucket has 300 tokens, it can handle a burst of 300 requests instantly.
- **Use Case**: Allows for both steady traffic and occasional bursts. Commonly used for scenarios where burst handling is required, like API rate limiting.

6. Concurrent Rate Limiting

- **Concept**: Limits the number of concurrent (simultaneous) requests or operations at any given moment, rather than over a time period.
- **Example**: Limit to 10 concurrent API requests. If there are already 10 active requests, additional requests must wait until one finishes.
- **Use Case**: Useful in scenarios where resource usage is a concern, such as limiting database connections or preventing service overloads.

7. Request Quotas

- **Concept**: Users or clients are assigned a fixed quota of requests they can make within a certain period (e.g., daily, weekly). Once the quota is exhausted, no more requests are allowed until the period resets.
- **Example**: 10,000 API requests per day. Once a user reaches this limit, they must wait until the next day for the quota to reset.
- **Use Case**: Common in APIs with usage tiers, ensuring fair use across users or clients.

8. Rate Limiting by IP, User, or API Key

- **Concept**: Rate limits are applied based on specific identifiers like IP address, user account, or API key. This allows different rate limits for different clients.
- **Example**: Limit 100 requests per minute per IP address or per API key.

- **Use Case**: Helps to enforce per-user or per-client limits, preventing abuse from specific users while not affecting others.

9. Dynamic Rate Limiting

- **Concept**: The rate limits are dynamically adjusted based on the current system load, user behavior, or other factors. This allows for more flexible and responsive rate limiting.
- **Example**: Increase rate limits during off-peak hours or reduce them if the system is under heavy load.
- **Use Case**: Useful in systems where traffic patterns vary significantly, and static rate limits may not be optimal.

Choosing the Right Technique

- **Fixed Window** is simple but can cause bursty behavior.
- **Sliding Window** is more precise and smoothes out bursts.
- **Leaky Bucket** ensures a steady rate of request processing.
- **Token Bucket** allows for burst handling while controlling the overall rate.
- **Concurrent Limiting** is ideal for controlling resource usage.
- **Request Quotas** are perfect for enforcing fair usage policies.

Each technique has its strengths and weaknesses, and the best choice depends on the specific requirements of the system you're designing.

4. Implementing Rate Limiting in .NET

Let's dive into the implementation of rate limiting in a .NET application. We'll cover a practical example of chained rate limiting using the Fixed Window technique and rate limiting by IP, user, or API key.

Chained Rate Limiting with Fixed Window

Chained rate limiting combines multiple rate limiting strategies to create a layered defense against excessive traffic. In this example, we'll chain two fixed window rate limiters: one by host (domain) and another by IP address.

Step 1: Install the Necessary NuGet Package

Start by installing the `System.Threading.RateLimiting` package:

```
dotnet add package System.Threading.RateLimiting
```

Step 2: Implement Chained Rate Limiting

Here's how you can set up chained rate limiting in your .NET application:

```
using Microsoft.AspNetCore.Http;
using System.Threading.RateLimiting;

public class RateLimitOptions
{
    public int PermitLimitInMinutes { get; set; } = 100;
    public int WindowInMinutes { get; set; } = 1;

    public int PermitLimitInHours { get; set; } = 1000;
    public int WindowInHours { get; set; } = 1;
}

public void ConfigureRateLimiting(RateLimitOptions options)
{
    var rateLimitOptions = new RateLimitOptions();

    // Chained Rate Limiter by Host and IP
    var globalLimiter = PartitionedRateLimiter.CreateChained(
        PartitionedRateLimiter.Create<HttpContext, string>(httpContext =>
            RateLimitPartition.GetFixedWindowLimiter(partitionKey:
            httpContext.Request.Headers.Host.ToString(),
            partition => new FixedWindowRateLimiterOptions
            {
                AutoReplenishment = true,
                PermitLimit = options.PermitLimitInMinutes,
                Window = TimeSpan.FromMinutes(options.WindowInMinutes)
            })),
        PartitionedRateLimiter.Create<HttpContext, string>(httpContext =>
            RateLimitPartition.GetFixedWindowLimiter(
            partitionKey: httpContext.Request.Headers["X-Forwarded-
For"].FirstOrDefault()
                          ??
httpContext.Request.HttpContext.Connection.RemoteIpAddress.MapToIPv4().ToS
tring(),
            partition => new FixedWindowRateLimiterOptions
            {
                AutoReplenishment = true,
                PermitLimit = options.PermitLimitInHours,
                Window = TimeSpan.FromHours(options.WindowInHours)
            }))
    );
```

```
}
```

Step 3: Applying Rate Limiting in Middleware

To apply the rate limiting configuration in your ASP.NET Core application, you can use middleware:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    // Use the rate limiting middleware
    app.Use(async (context, next) =>
    {
        var limiter =
context.RequestServices.GetRequiredService<PartitionedRateLimiter<HttpCont
ext>>();

        var lease = await limiter.AcquireAsync(context);

        if (lease.IsAcquired)
        {
            await next.Invoke();
        }
        else
        {
            context.Response.StatusCode =
StatusCodes.Status429TooManyRequests;
        }
    });

    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Rate Limiting by IP, User, or API Key

To implement rate limiting based on specific identifiers like IP address, user ID, or API key, follow these steps:

Step 1: Define a Rate Limiter by IP, User, or API Key

You can use a `PartitionedRateLimiter` to apply limits based on a unique identifier. For example:

```
public void ConfigureRateLimitingByUser()
{
    var userLimiter = PartitionedRateLimiter.Create<HttpContext, string>
(httpContext =>
        RateLimitPartition.GetFixedWindowLimiter(partitionKey:
httpContext.User.Identity.Name ?? "anonymous",
        partition => new FixedWindowRateLimiterOptions
        {
            AutoReplenishment = true,
            PermitLimit = 100,
            Window = TimeSpan.FromMinutes(1)
        }));
}
```

Step 2: Apply the Rate Limiting in Middleware

Just like with the chained rate limiter, apply this in middleware to enforce the limits based on user identity:

```
app.Use(async (context, next) =>
{
    var userLimiter =
context.RequestServices.GetRequiredService<PartitionedRateLimiter<HttpCont
ext>>();

    var lease = await userLimiter.AcquireAsync(context);

    if (lease.IsAcquired)
    {
        await next.Invoke();
    }
    else
    {
        context.Response.StatusCode =
StatusCodes.Status429TooManyRequests;
    }
});
```

5. AutoReplenishment in Rate Limiting

AutoReplenishment is a feature in rate limiting that automatically refills or replenishes the available permits or tokens at regular intervals, based on the defined time window, without requiring any manual intervention.

In the context of the `FixedWindowRateLimiter` or similar rate limiters:

- **Fixed Window Rate Limiter**: This type of rate limiter limits the number of actions (like API calls) that can occur within a fixed time window (e.g., 1 minute, 1 hour).

How AutoReplenishment Works:

- **Without AutoReplenishment**: When the permits (or allowed requests) are exhausted, they won't be refilled automatically. You'd either have to wait until the next fixed window starts, or manually reset the limiter.
- **With AutoReplenishment (AutoReplenishment = true)**: The permits are automatically refilled at the beginning of each new time window. For example, if you have a rate limit of 100

requests per minute, at the start of each new minute, the limiter automatically resets, allowing another 100 requests.

Example:

If you set a `FixedWindowRateLimiter` with `PermitLimit = 100` and `Window = TimeSpan.FromMinutes(1)`, and you enable `AutoReplenishment`:

- You can make 100 requests in that 1-minute window.
- After 1 minute, the limiter automatically resets, allowing another 100 requests in the next minute.

Purpose:

AutoReplenishment is useful for ensuring that rate limits are enforced in a consistent, predictable manner, without the need for manual resets or complex logic to handle window transitions. It simplifies the implementation of rate limiting by handling the reset mechanism internally.

6. Conclusion

Rate limiting is a critical feature for managing traffic to your APIs, ensuring fairness, and protecting your services from abuse. By understanding and implementing different rate limiting techniques like Fixed Window, Sliding Window, and Token Bucket, you can tailor your approach to suit your application's needs.

In this blog post, we've covered the core concepts of rate limiting and provided a practical implementation in .NET, including a chained rate limiting example and rate limiting based on IP, user, or API key. These techniques are essential tools in your arsenal to build resilient, secure, and efficient APIs.

By leveraging these rate limiting strategies, you can ensure that your API remains available and responsive, even under high traffic conditions, while also enforcing usage policies and protecting against malicious attacks.

Happy coding! 🚀

With this guide, you should have a solid foundation in rate limiting and be well-equipped to implement it

Article 12 / 5 May 2024 / .NET / 1 min read

Popular design patterns in .NET

The patterns that actually show up in .NET codebases, with when to reach for each and when not to.

Design patterns are essential tools in the arsenal of every .NET developer. They provide proven solutions to recurring design problems, promote code reusability, and improve maintainability. In this blog post, we'll explore three popular design patterns commonly used in .NET development: Singleton, Factory, and Observer.

Singleton Pattern

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. This pattern is useful when exactly one object is needed to coordinate actions across the system. Here's a basic implementation of the Singleton pattern in C#:

```
csharpCopy codepublic class Singleton
{
    private static Singleton instance;

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                instance = new Singleton();
            }
            return instance;
        }
    }

    public void SomeMethod()
    {
        Console.WriteLine("Singleton method called.");
    }
}
```

Usage:

```
csharpCopy codeSingleton.Instance.SomeMethod();
```

Factory Pattern

The Factory pattern is used to create objects without specifying the exact class of object that will be created. It defines an interface for creating objects, but allows subclasses to alter the type of objects that will be created. Here's a simple example of the Factory pattern:

```
csharpCopy codepublic interface IProduct
{
    void Operation();
}

public class ConcreteProductA : IProduct
{
    public void Operation()
    {
        Console.WriteLine("ConcreteProductA operation");
    }
}

public class ConcreteProductB : IProduct
{
    public void Operation()
    {
        Console.WriteLine("ConcreteProductB operation");
    }
}

public class ProductFactory
{
    public IProduct CreateProduct(string productType)
    {
        switch (productType)
        {
            case "A":
                return new ConcreteProductA();
            case "B":
                return new ConcreteProductB();
            default:
                throw new ArgumentException("Invalid product type");
        }
    }
}
```

Usage:

```
csharpCopy codeProductFactory factory = new ProductFactory();
IProduct productA = factory.CreateProduct("A");
IProduct productB = factory.CreateProduct("B");
productA.Operation();
productB.Operation();
```

Observer Pattern

The Observer pattern is a behavioral pattern where an object (the subject) maintains a list of its dependents (observers) and notifies them of any state changes. This pattern is commonly used in event handling systems. Here's a basic example of the Observer pattern:

```
csharpCopy codepublic interface IObserver
{
    void Update(string message);
}

public interface ISubject
{
    void Attach(IObserver observer);
    void Detach(IObserver observer);
    void Notify(string message);
}

public class ConcreteSubject : ISubject
{
    private List<IObserver> observers = new List<IObserver>();

    public void Attach(IObserver observer)
    {
        observers.Add(observer);
    }

    public void Detach(IObserver observer)
    {
        observers.Remove(observer);
    }

    public void Notify(string message)
    {
        foreach (var observer in observers)
        {
            observer.Update(message);
        }
    }

    public void DoSomething()
    {
        // Do something and notify observers
        Notify("Something happened");
    }
}
```

```
public class ConcreteObserver : IObserver
{
    public void Update(string message)
    {
        Console.WriteLine($"Received message: {message}");
    }
}
```

Usage:

```
csharpCopy codeConcreteSubject subject = new ConcreteSubject();
ConcreteObserver observerA = new ConcreteObserver();
ConcreteObserver observerB = new ConcreteObserver();

subject.Attach(observerA);
subject.Attach(observerB);

subject.DoSomething();
```

In conclusion, understanding and applying design patterns in your .NET projects can lead to more maintainable, scalable, and flexible code. The Singleton, Factory, and Observer patterns are just a few examples of the many design patterns available to .NET developers, each with its own use cases and benefits. By incorporating these patterns into your designs, you can write cleaner, more efficient code that is easier to maintain and extend.

Article 13 / 8 Mar 2023 / .NET / 2 min read

Dynamic objects in C#

What dynamic and ExpandoObject really do at runtime, and the narrow cases where they earn their place.

---

n C#, a dynamic object is a type that is resolved at runtime rather than at compile-time. This means that the type of a dynamic object is not known until the code is executed, which allows you to use the object to perform operations that would otherwise not be possible with statically-typed objects.

For example, consider the following code:

```
codedynamic myDynamicObject = "hello";
myDynamicObject = 123;
myDynamicObject = new { Name = "John", Age = 30 };

Console.WriteLine(myDynamicObject.Name);
Console.WriteLine(myDynamicObject.Age);
```

In this code, we declare a variable called `myDynamicObject` and assign it a string, an integer, and then an anonymous object. Because `myDynamicObject` is a dynamic object, we can change its type at runtime.

Later in the code, we try to access the `Name` and `Age` properties of `myDynamicObject`. Because the type of `myDynamicObject` is not known until runtime, the code will compile without error even though the properties `Name` and `Age` are not defined on the first two types we assigned to `myDynamicObject`.

When the code is executed, however, the third assignment to `myDynamicObject` creates an anonymous object that has properties called `Name` and `Age`, and so the code will output the values of those properties to the console.

Dynamic objects can be useful in certain situations where you need to work with data that has unknown or varying types at runtime. However, because dynamic objects do not provide compile-time type checking, they can also introduce additional runtime errors that would have

been caught at compile-time with statically-typed objects. It's important to use dynamic objects judiciously and only when necessary.

How Dynamic Objects Work in C#: An Example

When the `myDynamicObject` is assigned an anonymous object, the previously assigned string and integer values are lost and are no longer accessible.

This is because `myDynamicObject` is a dynamic object, which means its type can be changed at runtime. When you assign an anonymous object to `myDynamicObject`, the type of `myDynamicObject` changes to the type of the anonymous object, and the previous values assigned to `myDynamicObject` are no longer available.

So in the code snippet from the previous question, after the line `myDynamicObject = new { Name = "John", Age = 30 };`, the string and integer values assigned to `myDynamicObject` are lost and cannot be accessed again. Only the `Name` and `Age` properties of the anonymous object assigned to `myDynamicObject` can be accessed.

Article 14 / 8 Mar 2023 / .NET / 1 min read

ViewBag, ViewData and TempData in ASP.NET MVC

Three ways to pass data to a view, how their lifetimes differ, and which one to use.

`ViewBag`, `ViewData`, and `TempData` are all mechanisms provided by ASP.NET MVC to share data between controller actions and views.

`ViewBag` and `ViewData` are similar in that they both allow you to pass data from the controller to the view, but they differ in their underlying implementation.

`ViewBag` is a dynamic object that uses C# dynamic features to store and retrieve data. You can set values on the `ViewBag` object in the controller and then access those values in the view using dot notation. For example, in the controller, you might set a value like this:

```
ViewBag.Title = "My Page Title";
```

And then in the view, you could access that value like this:

```
code<title>@ViewBag.Title</title>
```

`ViewData`, on the other hand, uses a dictionary to store and retrieve data. You set values on the `ViewData` dictionary in the controller, and then you can access those values in the view using the `["key"]` syntax. For example, in the controller, you might set a value like this:

```
 codeViewData["Title"] = "My Page Title";
```

And then in the view, you could access that value like this:

```
code<title>@ViewData["Title"]</title>
```

The `TempData` dictionary is similar to `ViewData`, but it is used to pass data between actions during a single user request. When you set a value in `TempData` in one action, you can retrieve that value in a subsequent action. However, once the request is complete, the `TempData` values are discarded.

In summary, `ViewBag` and `ViewData` are both used to pass data from a controller to a view, but `ViewBag` is a dynamic object, and `ViewData` is a dictionary. `TempData` is used to pass data between actions during a single user request

Part V

Places

Every place I've lived, worked or travelled to. Mostly the Kathmandu valley so far, so the map has plenty of room.

2 places in 1 country so far

Kathmandu

home, 27.72°N 85.32°E

Where I live, and where I work now at Himalayan Everest Insurance.

Lalitpur

2022 - 2024, 27.66°N 85.32°E

Across the Bagmati: NBIT, then EKbana Solutions, where I built insurance backends for two and a half years.

Contact

Got a system that needs a steady hand?

Open to senior and lead roles in backend architecture and .NET modernisation, in Kathmandu or remote.

sthaejan00@gmail.com

+977 986 213 5998 / ejanshrestha.com.np / linkedin.com/in/ejanshrestha