Host ASP.NET Core on Linux with Nginx

Article • 10/07/2023

By Sourabh Shirhatti

This guide explains setting up a production-ready ASP.NET Core environment for Ubuntu, Red Hat Enterprise (RHEL), and SUSE Linux Enterprise Server.

For information on other Linux distributions supported by ASP.NET Core, see Prerequisites for .NET Core on Linux.

This guide:

- Places an existing ASP.NET Core app behind a reverse proxy server.
- Sets up the reverse proxy server to forward requests to the Kestrel web server.
- Ensures the web app runs on startup as a daemon.
- Configures a process management tool to help restart the web app.

Prerequisites

Ubuntu

- Access to Ubuntu 20.04 with a standard user account with sudo privilege.
- The latest stable .NET runtime installed on the server.
- An existing ASP.NET Core app.

At any point in the future after upgrading the shared framework, restart the ASP.NET Core apps hosted by the server.

Publish and copy over the app

Configure the app for a framework-dependent deployment.

If the app is run locally in the Development environment and isn't configured by the server to make secure HTTPS connections, adopt either of the following approaches:

- Configure the app to handle secure local connections. For more information, see the HTTPS configuration section.
- Configure the app to run at the insecure endpoint:

- Deactivate HTTPS Redirection Middleware in the Development environment (`Program.cs`):

  C#

  ```
  if (!app.Environment.IsDevelopment())
  {
      app.UseHttpsRedirection();
  }
  ```

  For more information, see Use multiple environments in ASP.NET Core.

- Remove `https://localhost:5001` (if present) from the `applicationUrl` property in the `Properties/launchSettings.json` file.

For more information on configuration by environment, see Use multiple environments in ASP.NET Core.

Run dotnet publish from the development environment to package an app into a directory (for example, `bin/Release/{TARGET FRAMEWORK MONIKER}/publish`, where the `{TARGET FRAMEWORK MONIKER}` placeholder is the Target Framework Moniker (TFM)) that can run on the server:

.NET CLI

```
dotnet publish --configuration Release
```

The app can also be published as a self-contained deployment if you prefer not to maintain the .NET Core runtime on the server.

Copy the ASP.NET Core app to the server using a tool that integrates into the organization's workflow (for example, `SCP`, `SFTP`). It's common to locate web apps under the `var` directory (for example, `var/www/helloapp`).

> **Note**
>
> Under a production deployment scenario, a continuous integration workflow does the work of publishing the app and copying the assets to the server.

Test the app:

1. From the command line, run the app: `dotnet <app_assembly>.dll`.

2. In a browser, navigate to `http://<serveraddress>:<port>` to verify the app works on Linux locally.

Configure a reverse proxy server

A reverse proxy is a common setup for serving dynamic web apps. A reverse proxy terminates the HTTP request and forwards it to the ASP.NET Core app.

Use a reverse proxy server

Kestrel is great for serving dynamic content from ASP.NET Core. However, the web serving capabilities aren't as feature rich as servers such as IIS, Apache, or Nginx. A reverse proxy server can offload work such as serving static content, caching requests, compressing requests, and HTTPS termination from the HTTP server. A reverse proxy server may reside on a dedicated machine or may be deployed alongside an HTTP server.

For the purposes of this guide, a single instance of Nginx is used. It runs on the same server, alongside the HTTP server. Based on requirements, a different setup may be chosen.

Because requests are forwarded by reverse proxy, use the Forwarded Headers Middleware from the Microsoft.AspNetCore.HttpOverrides package, which is automatically included in ASP.NET Core apps via the shared framework's Microsoft.AspNetCore.App metapackage. The middleware updates the `Request.Scheme`, using the `X-Forwarded-Proto` header, so that redirect URIs and other security policies work correctly.

Forwarded Headers Middleware should run before other middleware. This ordering ensures that the middleware relying on forwarded headers information can consume the header values for processing. To run Forwarded Headers Middleware after diagnostics and error handling middleware, see Forwarded Headers Middleware order.

Invoke the UseForwardedHeaders method before calling other middleware. Configure the middleware to forward the `X-Forwarded-For` and `X-Forwarded-Proto` headers:

C#

```
using Microsoft.AspNetCore.HttpOverrides;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication();
```

```
var app = builder.Build();

app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor |
ForwardedHeaders.XForwardedProto
});

app.UseAuthentication();

app.MapGet("/", () => "Hello ForwardedHeadersOptions!");

app.Run();
```

If no ForwardedHeadersOptions are specified to the middleware, the default headers to forward are `None`.

Proxies running on loopback addresses (`127.0.0.0/8`, `[::1]`), including the standard localhost address (`127.0.0.1`), are trusted by default. If other trusted proxies or networks within the organization handle requests between the internet and the web server, add them to the list of KnownProxies or KnownNetworks with ForwardedHeadersOptions. The following example adds a trusted proxy server at IP address `10.0.0.100` to the Forwarded Headers Middleware `KnownProxies`:

C#

```
using Microsoft.AspNetCore.HttpOverrides;
using System.Net;

var builder = WebApplication.CreateBuilder(args);

// Configure forwarded headers
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.KnownProxies.Add(IPAddress.Parse("10.0.0.100"));
});

builder.Services.AddAuthentication();

var app = builder.Build();

app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor |
ForwardedHeaders.XForwardedProto
});

app.UseAuthentication();
```

```
app.MapGet("/", () => "10.0.0.100");

app.Run();
```

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Install Nginx

Ubuntu

Use `apt-get` to install Nginx. The installer creates a systemd init script that runs Nginx as daemon on system startup. Follow the installation instructions for Ubuntu at Nginx: Official Debian/Ubuntu packages.

> **Note**
>
> If optional Nginx modules are required, building Nginx from source might be required.

Since Nginx was installed for the first time, explicitly start it by running:

Bash

```
sudo service nginx start
```

Verify a browser displays the default landing page for Nginx. The landing page is reachable at `http://<server_IP_address>/index.nginx-debian.html`.

Configure Nginx

Ubuntu

To configure Nginx as a reverse proxy to forward HTTP requests to the ASP.NET Core app, modify `/etc/nginx/sites-available/default` and recreate the symlink. After creating the `/etc/nginx/sites-available/default` file, use the following command to create the symlink:

Bash

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

Open `/etc/nginx/sites-available/default` in a text editor, and replace the contents with the following snippet:

text

```
http {
  map $http_connection $connection_upgrade {
    "~*Upgrade" $http_connection;
    default keep-alive;
  }

  server {
    listen        80;
    server_name   example.com *.example.com;
    location / {
        proxy_pass         http://127.0.0.1:5000/;
        proxy_http_version 1.1;
        proxy_set_header   Upgrade $http_upgrade;
        proxy_set_header   Connection $connection_upgrade;
        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 $scheme;
    }
  }
}
```

If the app is a SignalR or Blazor Server app, see ASP.NET Core SignalR production hosting and scaling and Host and deploy ASP.NET Core server-side Blazor apps respectively for more information.

When no `server_name` matches, Nginx uses the default server. If no default server is defined, the first server in the configuration file is the default server. As a best practice, add a specific default server that returns a status code of 444 in your configuration file. A default server configuration example is:

text

```
server {
    listen   80 default_server;
    # listen [::]:80 default_server deferred;
```

```
    return   444;
}
```

With the preceding configuration file and default server, Nginx accepts public traffic on port 80 with host header `example.com` or `*.example.com`. Requests not matching these hosts won't get forwarded to Kestrel. Nginx forwards the matching requests to Kestrel at `http://127.0.0.1:5000/`. For more information, see How nginx processes a request. To change Kestrel's IP/port, see Kestrel: Endpoint configuration.

> ⚠ **Warning**
>
> Failure to specify a proper **server_name directive** exposes your app to security vulnerabilities. Subdomain wildcard binding (for example, `*.example.com`) doesn't pose this security risk if you control the entire parent domain (as opposed to `*.com`, which is vulnerable). For more information, see **RFC 9110: HTTP Semantics (Section 7.2: Host and :authority)**.

Once the Nginx configuration is established, run `sudo nginx -t` to verify the syntax of the configuration files. If the configuration file test is successful, force Nginx to pick up the changes by running `sudo nginx -s reload`.

To directly run the app on the server:

1. Navigate to the app's directory.
2. Run the app: `dotnet <app_assembly.dll>`, where `app_assembly.dll` is the assembly file name of the app.

Ubuntu

If the app runs on the server but fails to respond over the internet, check the server's firewall and confirm port 80 is open. If using an Azure Ubuntu VM, add a Network Security Group (NSG) rule that enables inbound port 80 traffic. There's no need to enable an outbound port 80 rule, as the outbound traffic is automatically granted when the inbound rule is enabled.

When done testing the app, shut down the app with `Ctrl`+`C` (Windows) or `⌘`+`C` (macOS) at the command prompt.

Increase keepalive_requests

keepalive_requests can be increased for higher performance, For more information, see this GitHub issue.

Monitor the app

The server is set up to forward requests made to `http://<serveraddress>:80` on to the ASP.NET Core app running on Kestrel at `http://127.0.0.1:5000`. However, Nginx isn't set up to manage the Kestrel process. systemd can be used to create a service file to start and monitor the underlying web app. `systemd` is an init system that provides many powerful features for starting, stopping, and managing processes.

Create the service file

Create the service definition file:

Bash

```
sudo nano /etc/systemd/system/kestrel-helloapp.service
```

The following example is an `.ini` service file for the app:

text

```
[Unit]
Description=Example .NET Web API App running on Linux

[Service]
WorkingDirectory=/var/www/helloapp
ExecStart=/usr/bin/dotnet /var/www/helloapp/helloapp.dll
Restart=always
# Restart service after 10 seconds if the dotnet service crashes:
RestartSec=10
KillSignal=SIGINT
SyslogIdentifier=dotnet-example
User=www-data
Environment=ASPNETCORE_ENVIRONMENT=Production
Environment=DOTNET_NOLOGO=true

[Install]
WantedBy=multi-user.target
```

In the preceding example, the user that manages the service is specified by the `User` option. The user (`www-data`) must exist and have proper ownership of the app's files.

Use `TimeoutStopSec` to configure the duration of time to wait for the app to shut down after it receives the initial interrupt signal. If the app doesn't shut down in this period, SIGKILL is issued to terminate the app. Provide the value as unitless seconds (for example, `150`), a time span value (for example, `2min 30s`), or `infinity` to disable the timeout. `TimeoutStopSec` defaults to the value of `DefaultTimeoutStopSec` in the manager configuration file (`systemd-system.conf`, `system.conf.d`, `systemd-user.conf`, `user.conf.d`). The default timeout for most distributions is 90 seconds.

text

```
# The default value is 90 seconds for most distributions.
TimeoutStopSec=90
```

Linux has a case-sensitive file system. Setting `ASPNETCORE_ENVIRONMENT` to `Production` results in searching for the configuration file `appsettings.Production.json`, not `appsettings.production.json`.

Some values (for example, SQL connection strings) must be escaped for the configuration providers to read the environment variables. Use the following command to generate a properly escaped value for use in the configuration file:

Console

```
systemd-escape "<value-to-escape>"
```

Colon (`:`) separators aren't supported in environment variable names. Use a double underscore (`__`) in place of a colon. The Environment Variables configuration provider converts double-underscores into colons when environment variables are read into configuration. In the following example, the connection string key `ConnectionStrings:DefaultConnection` is set into the service definition file as `ConnectionStrings__DefaultConnection`:

Console

```
Environment=ConnectionStrings__DefaultConnection={Connection String}
```

Save the file and enable the service.

Bash

```
sudo systemctl enable kestrel-helloapp.service
```

Start the service and verify that it's running.

Bash

```
sudo systemctl start kestrel-helloapp.service
sudo systemctl status kestrel-helloapp.service

● kestrel-helloapp.service - Example .NET Web API App running on
Linux
    Loaded: loaded (/etc/systemd/system/kestrel-helloapp.service; en-
abled)
    Active: active (running) since Thu 2016-10-18 04:09:35 NZDT; 35s
ago
Main PID: 9021 (dotnet)
    CGroup: /system.slice/kestrel-helloapp.service
            └─9021 /usr/local/bin/dotnet
/var/www/helloapp/helloapp.dll
```

With the reverse proxy configured and Kestrel managed through `systemd`, the web app is fully configured and can be accessed from a browser on the local machine at `http://localhost`. It's also accessible from a remote machine, barring any firewall that might be blocking. Inspecting the response headers, the `Server` header shows the ASP.NET Core app being served by Kestrel.

text

```
HTTP/1.1 200 OK
Date: Tue, 11 Oct 2016 16:22:23 GMT
Server: Kestrel
Keep-Alive: timeout=5, max=98
Connection: Keep-Alive
Transfer-Encoding: chunked
```

View logs

Since the web app using Kestrel is managed using systemd, all events and processes are logged to a centralized journal. However, this journal includes all entries for all services and processes managed by `systemd`. To view the `kestrel-helloapp.service`-specific items, use the following command:

Bash

```
sudo journalctl -fu kestrel-helloapp.service
```

For further filtering, time options such as `--since today`, `--until 1 hour ago`, or a combination of these can reduce the number of entries returned.

Bash

```
sudo journalctl -fu kestrel-helloapp.service --since "2016-10-18" --
until "2016-10-18 04:00"
```

Data protection

The ASP.NET Core Data Protection stack is used by several ASP.NET Core middlewares, including authentication middleware (for example, cookie middleware) and cross-site request forgery (CSRF) protections. Even if Data Protection APIs aren't called by user code, data protection should be configured to create a persistent cryptographic key store. If data protection isn't configured, the keys are held in memory and discarded when the app restarts.

If the key ring is stored in memory when the app restarts:

- All cookie-based authentication tokens are invalidated.
- Users are required to sign in again on their next request.
- Any data protected with the key ring can no longer be decrypted. This may include CSRF tokens and ASP.NET Core MVC TempData cookies.

To configure data protection to persist and encrypt the key ring, see:

- Key storage providers in ASP.NET Core
- Key encryption at rest in Windows and Azure using ASP.NET Core

Long request header fields

Proxy server default settings typically limit request header fields to 4 K or 8 K depending on the platform. An app may require fields longer than the default (for example, apps that use Microsoft Entra ID). If longer fields are required, the proxy server's default settings require adjustment. The values to apply depend on the scenario. For more information, see your server's documentation.

- proxy_buffer_size
- proxy_buffers
- proxy_busy_buffers_size
- large_client_header_buffers

> ⚠ **Warning**
>
> Don't increase the default values of proxy buffers unless necessary. Increasing these values increases the risk of buffer overrun (overflow) and Denial of Service (DoS) attacks by malicious users.

Secure the app

Enable AppArmor

Linux Security Modules (LSM) is a framework that's part of the Linux kernel since Linux 2.6. LSM supports different implementations of security modules. AppArmor is an LSM that implements a Mandatory Access Control system, which allows confining the program to a limited set of resources. Ensure AppArmor is enabled and properly configured.

Configure the firewall

Close off all external ports that aren't in use. Uncomplicated firewall (ufw) provides a front end for `iptables` by providing a CLI for configuring the firewall.

Ubuntu

> ⚠ **Warning**
>
> A firewall prevents access to the whole system if not configured correctly. Failure to specify the correct SSH port effectively locks you out of the system if you are using SSH to connect to it. The default port is 22. For more information, see the **introduction to ufw** and the **manual**.

Install `ufw` and configure it to allow traffic on any ports needed.

Bash

```
sudo apt-get install ufw

sudo ufw allow 22/tcp
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp
```

```
sudo ufw enable
```

Secure Nginx

Change the Nginx response name

Edit `src/http/ngx_http_header_filter_module.c`:

```
C

static char ngx_http_server_string[] = "Server: Web Server" CRLF;
static char ngx_http_server_full_string[] = "Server: Web Server"
CRLF;
```

Configure options

Configure the server with additional required modules. Consider using a web app firewall, such as ModSecurity, to harden the app.

HTTPS configuration

Configure the app for secure (HTTPS) local connections

The dotnet run command uses the app's `Properties/launchSettings.json` file, which configures the app to listen on the URLs provided by the `applicationUrl` property. For example, `https://localhost:5001;http://localhost:5000`.

Configure the app to use a certificate in development for the `dotnet run` command or development environment (`F5` or `Ctrl`+`F5` in Visual Studio Code) using one of the following approaches:

- Replace the default certificate from configuration (*Recommended*)
- KestrelServerOptions.ConfigureHttpsDefaults

Configure the reverse proxy for secure (HTTPS) client connections

> ⚠ **Warning**
>
> The security configuration in this section is a general configuration to be used as a starting point for further customization. We're unable to provide support for third-

party tooling, servers, and operating systems. *Use the configuration in this section at your own risk.* For more information, access the following resources:

- **Configuring HTTPS servers** (Nginx documentation)
- **mozilla.org SSL Configuration Generator**

- Configure the server to listen to HTTPS traffic on port 443 by specifying a valid certificate issued by a trusted Certificate Authority (CA).
- Harden the security by employing some of the practices depicted in the following */etc/nginx/nginx.conf* file.
- The following example doesn't configure the server to redirect insecure requests. We recommend using HTTPS Redirection Middleware. For more information, see Enforce HTTPS in ASP.NET Core.

  > **Note**
  >
  > For development environments where the server configuration handles secure redirection instead of HTTPS Redirection Middleware, we recommend using temporary redirects (302) rather than permanent redirects (301). Link caching can cause unstable behavior in development environments.

- Adding a `Strict-Transport-Security` (HSTS) header ensures all subsequent requests made by the client are over HTTPS. For guidance on setting the `Strict-Transport-Security` header, see Enforce HTTPS in ASP.NET Core.
- If HTTPS will be disabled in the future, use one of the following approaches:
  - Don't add the HSTS header.
  - Choose a short `max-age` value.

Add the */etc/nginx/proxy.conf* configuration file:

nginx

```
proxy_redirect          off;
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;
client_max_body_size    10m;
client_body_buffer_size 128k;
proxy_connect_timeout   90;
proxy_send_timeout      90;
```

```
proxy_read_timeout      90;
proxy_buffers           32 4k;
```

Ubuntu

Replace the contents of the */etc/nginx/nginx.conf* configuration file with the following file. The example contains both `http` and `server` sections in one configuration file.

nginx

```
http {
    include        /etc/nginx/proxy.conf;
    limit_req_zone $binary_remote_addr zone=one:10m rate=5r/s;
    server_tokens  off;

    sendfile on;
    # Adjust keepalive_timeout to the lowest possible value that
makes sense
    # for your use case.
    keepalive_timeout   29;
    client_body_timeout 10; client_header_timeout 10; send_timeout
10;

    upstream helloapp{
        server 127.0.0.1:5000;
    }

    server {
        listen                  443 ssl http2;
        listen                  [::]:443 ssl http2;
        server_name             example.com *.example.com;
        ssl_certificate         /etc/ssl/certs/testCert.crt;
        ssl_certificate_key     /etc/ssl/certs/testCert.key;
        ssl_session_timeout     1d;
        ssl_protocols           TLSv1.2 TLSv1.3;
        ssl_prefer_server_ciphers off;
        ssl_ciphers             ECDHE-ECDSA-AES128-GCM-
SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-
SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-CHACHA20-
POLY1305:ECDHE-RSA-CHACHA20-POLY1305:DHE-RSA-AES128-GCM-SHA256:DHE-
RSA-AES256-GCM-SHA384;
        ssl_session_cache       shared:SSL:10m;
        ssl_session_tickets     off;
        ssl_stapling            off;

        add_header X-Frame-Options DENY;
        add_header X-Content-Type-Options nosniff;

        #Redirects all traffic
```

```
        location / {
            proxy_pass http://helloapp;
            limit_req  zone=one burst=10 nodelay;
        }
    }
}
```

> **Note**
>
> Blazor WebAssembly apps require a larger `burst` parameter value to accommodate the larger number of requests made by an app. For more information, see **Host and deploy ASP.NET Core Blazor WebAssembly**.

> **Note**
>
> The preceding example disables Online Certificate Status Protocol (OCSP) Stapling. If enabled, confirm that the certificate supports the feature. For more information and guidance on enabling OCSP, see the following properties in the **Module ngx_http_ssl_module (Nginx documentation)** article:
>
> - `ssl_stapling`
> - `ssl_stapling_file`
> - `ssl_stapling_responder`
> - `ssl_stapling_verify`

Secure Nginx from clickjacking

Clickjacking, also known as a *UI redress attack*, is a malicious attack where a website visitor is tricked into clicking a link or button on a different page than they're currently visiting. Use `X-FRAME-OPTIONS` to secure the site.

To mitigate clickjacking attacks:

1. Edit the *nginx.conf* file:

   Bash

   ```
   sudo nano /etc/nginx/nginx.conf
   ```

   Within the `http{}` code block, add the line: `add_header X-Frame-Options "SAMEORIGIN";`

2. Save the file.

3. Restart Nginx.

MIME-type sniffing

This header prevents most browsers from MIME-sniffing a response away from the declared content type, as the header instructs the browser not to override the response content type. With the `nosniff` option, if the server says the content is `text/html`, the browser renders it as `text/html`.

1. Edit the *nginx.conf* file:

   ```
   Bash

   sudo nano /etc/nginx/nginx.conf
   ```

   Within the `http{}` code block, add the line: `add_header X-Content-Type-Options "nosniff";`

2. Save the file.

3. Restart Nginx.

Additional Nginx suggestions

After upgrading the shared framework on the server, restart the ASP.NET Core apps hosted by the server.

Additional resources

- Prerequisites for .NET Core on Linux
- Nginx: Binary Releases: Official Debian/Ubuntu packages
- Troubleshoot and debug ASP.NET Core projects
- Configure ASP.NET Core to work with proxy servers and load balancers
- NGINX: Using the Forwarded header

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback