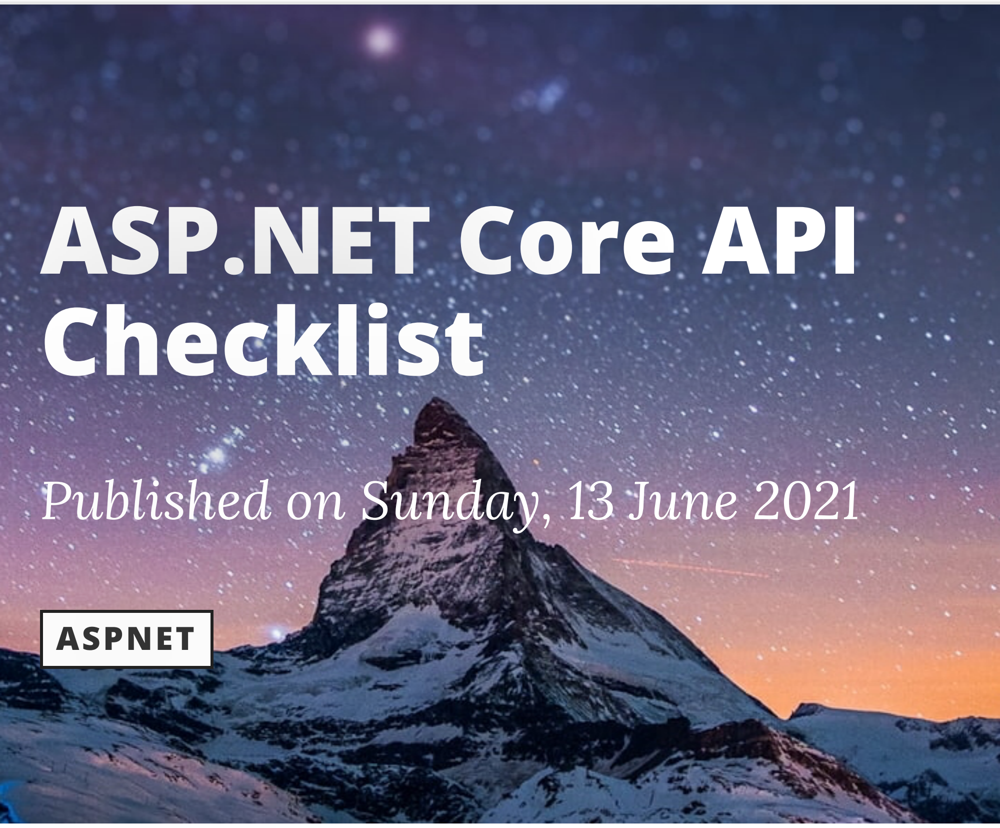

ASP.NET Core API Checklist

Published on Sunday, 13 June 2021

ASPNET

Building modern APIs require a lot of things to make them reliable, observable, and scalable. In no particular order, here are some of them that help you build better APIs:

1. Healthchecks

Healthchecks are important in making sure that we know when anything happens to our APIs. We can setup dashboards to monitor them and setup alerting to let us know when one of the APIs is unhealthy. They are also important when deploying your apps to kubernetes. Kubernetes can monitor healthchecks of your APIs and automatically try to kill the unhealthy instances and create new instances to take their place.

There are two kinds of healthchecks:

- **Liveliness:** indicates if your API has crashed and must be restarted.

- **Readiness:** indicates if your API has been intialized and is ready for processing requests. When launching a new instance of an

API, it might need some time to intialize dependencies and load data to be ready.

More Information:

- MS Docs - Health checks in ASP.NET Core
- IAmTimCorey - Intro to Health Checks in .NET Core

2. Logging

Logging provides valuable information when trying to debug unexpected behavior. But too much logging can significantly slow down our APIs. For that reason we can set the logging level to `Warning` in production and only lower it when we need to.

By default ASP.NET Core provides an abstraction layer for logging that supports Structured Logging.

A very popular logging library that many people use with ASP.NET Core is Serilog. Serilog has more Sinks than the default ASP.NET Core loggging abstraction and can easily be integrated with ASP.NET Core.

More information:

- MS Docs - Logging in .NET Core and ASP.NET Core
- IAmTimCorey - C# Logging with Serilog and Seq - Structured Logging Made Easy

3. Observability

This includes a few things: 2. Performance monitoring (P99, P95, P50) 3. Metrics: Specific counters you or your business cares about 4. Tracing: Being able to see the entire lifecycle of each request from frontend to API to data source.

OpenTelemetry is an open standard for doing all of the above and ASP.NET Core supports it. The good news is, if you use OpenTelemetry, there is a rich ecosystem of tools and services that you can integrate with.

All of the major cloud providers have services that you can use to view the captured data.

4. Error reporting

There are tools specifically for capturing, storing and showing exceptions that have been raised in your APIs. They the exceptions by their type and location and show how many times they have occurred. Some tools include:

- Sentry
- Rollbar

- Raygun

5. Status Endpoint

It's also good to have a status endpoint in your API that shows the name of the API, version of the API and when was it started. This can be used to create a dashboard showing all of the different services and their versions. Something like this:

```
using
Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

using System;
using System.Diagnostics;

namespace MyCoolApi.Controllers
{
    public class StatusResponse
    {
        public string Name { get; set;
}
        public string Version { get;
set; }
        public DateTime StartTime {
get; set; }
        public string Host { get; set;
}
```

```
    }

    [ApiController]
    [Route("status")]
    public class StatusController
    {
        [HttpGet]
        public StatusResponse Get()
        {
            var version = 
typeof(Startup).Assembly.GetName().Versi


            return new StatusResponse
            {
                Name = "my-cool-api",
                Version = $"
{version.Major}.{version.Minor}.
{version.Build}",
                StartTime = 
Process.GetCurrentProcess().StartTime,
                Host = 
Environment.MachineName
            };
        }
    }
}
```

6. Http Resiliency

Although it's generally preferred for your APIs to communicate with other APIs using asynchronous messaging, sometimes you need to call other APIs using HTTP calls.

We need to bake in some level of resiliency by automatically retrying transient failures. This can easily be done by using something like Polly.

More information:

- Implement HTTP call retries with exponential backoff with IHttpClientFactory and Polly policies
- Bryan Hogan - Fault Tolerant Web Service Requests with Polly

7. Statelessness and Containerization

Containers are a great way to make sure your APIs can be easily scaled out and deployed to multiple environments in a repeatable manner. However to make sure you can get the most out of containerization you should try to make sure your APIs are stateless.

Being stateless means that they don't hold any critical data in memory. For caching you can use a centralized caching technology like Redis instead. This way you can start as many instances as you need without worrying about having stale cached data or data duplication.

You must also be careful about background jobs. You must make sure the different instances don't process background jobs multiple times. And for message queues, you have to implement the Competing Consumers pattern which some message buses support natively.

More information:

- MS Docs - Tutorial:

Containerize a .NET Core app
- Hangfire
- Quartz.NET
- MS Docs - Competing Consumers

9. OpenAPI Spec / Swagger

Documenting your APIs is very important. Swagger integrates with ASP.NET Core and automatically finds all of the routes (Controller actions) and shows them in a beautiful dashboard that you can customize.

More information:

- MS Docs - ASP.NET Core web API documentation with Swagger / OpenAPI
- Kevin Dockx - Documenting an ASP.NET Core API with OpenAPI / Swagger

10. Configuration and Options

ASP.NET Core has an extensible configuration mechanism. It can pull configurations from json files, environment variables and command line arguments. You can also provide custom sources for configuration. It also provide ways to easily fetch the configurations in a type-safe manner. it also provides an easy mechanism to validate the configuration sections.

More information:

- MS Docs - Options Pattern in ASP.NET Core
- Steve Gordon - Using Configuration and Options in .NET Core and ASP.NET Core Apps

11. Integration and

Unit Tests

ASP.NET Core has made it easy to write Unit tests by making the whole framework DI friendly. It has also made Integration tests easy by `WebApplicationFactory` Having automated tests saves a lot of time and makes your APIs more robust. And when writing integration tests, try to use the same database technology that you use for production. If you're using Postgres in production, don't use Sqlite or In-Memory DB Providers for integration tests.

More information:

- MS Docs - Integration tests in ASP.NET Core
- Steve Gordon - Integration Testing ASP.NET Core Applications: Best Practices

12. Build beautiful REST APIs

If you're building REST APIs, there are some conventions that make your APIs more pleasant and intuitive to use.

More information:

- Stackoverflow Blog - Best practices for REST API design

13. Authentication and Authorization

Authentication is the process of identifying a user and authorization is knowing and enforcing what each user can and can't do. The most popular standards for authentication is OpenIDConnect which is an authentication layer on top of OAuth 2.

There are some popular Identity Providers that you can easily integrate with your API:

- Auth0
- Okta
- FusionAuth

And there are some open source Identity and Access Management servers that you can run on-prem:

- Keycloak
- Gluu

And there are some libraries that you can use to build your own OIDC server:

- IdentityServer
- OpenIddict

More information

- Kevin Dockx - Securing ASP.NET Core 3 with OAuth2 and OpenID

Connect
- Kevin Dockx - Securing Microservices in ASP.NET Core

14. Security

14.1 CORS

Cross-Origin Resource Sharing allows frontends to call your API even if they are not on the same domain as the API. By default in ASP.NET Core CORS is disabled.

More information

- MS Docs - Enable Cross-Origin Requests (CORS) in ASP.NET Core

14.2 HTTPS Enforcing

For this there are two scenarios:

- You're using Kestrel on edge: Then you have to make sure it's only listening to and respond

over HTTPS.
- You've put ASP.NET Core behind a reverse proxy: Then you generally terminate HTTPS on the proxy and it's the proxy's job to enforce HTTPS.

More Information

- MS Docs - Enforce HTTPS in ASP.NET Core

14.3 Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is a security vulnerability which enables an attacker to place client side scripts (usually JavaScript) into web pagesMore Information. You can prevent it by sanitizing inputs from the user.

- Prevent Cross-Site Scripting (XSS) in ASP.NET Core

15. API Versioning

Versioning your APIs allow you to maintain backward compatibility when making breaking changes. You can maintaing multiple versions at the same time and then deprecate versions over time.

More Information

- Overview of API Versioning in ASP.NET Core 3.0+

Updates:

- Fixed ordering
- Added 13 (Auth) and 14 (Security). Special thanks to Matti-Koopa.
- Added 15 (API Versioning)