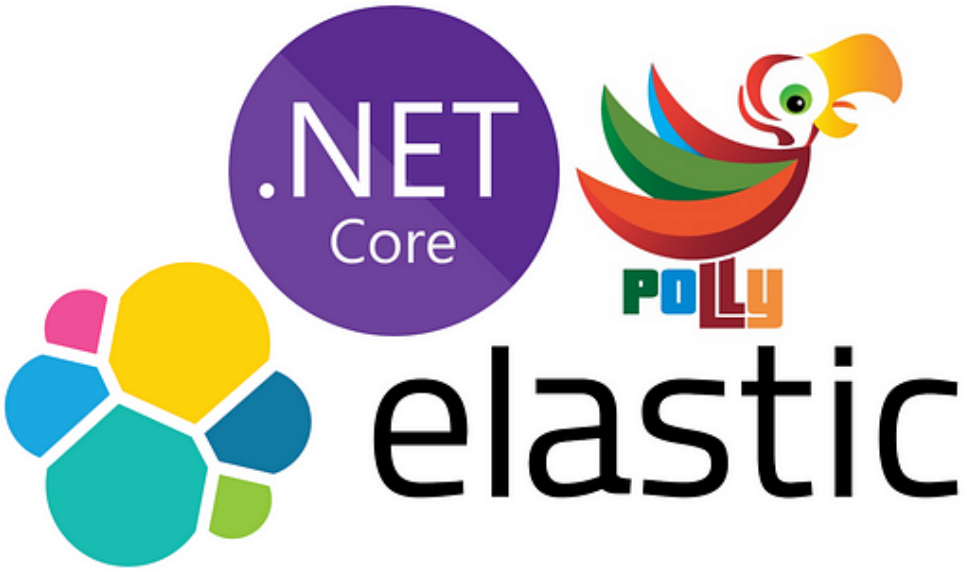
Microservices Observability, Resilience, Monitoring on .Net

Building Cross-Cutting Concerns — Microservices Observability with Distributed Logging, Health Monitoring, Resilient and Fault Tolerance with using Polly.



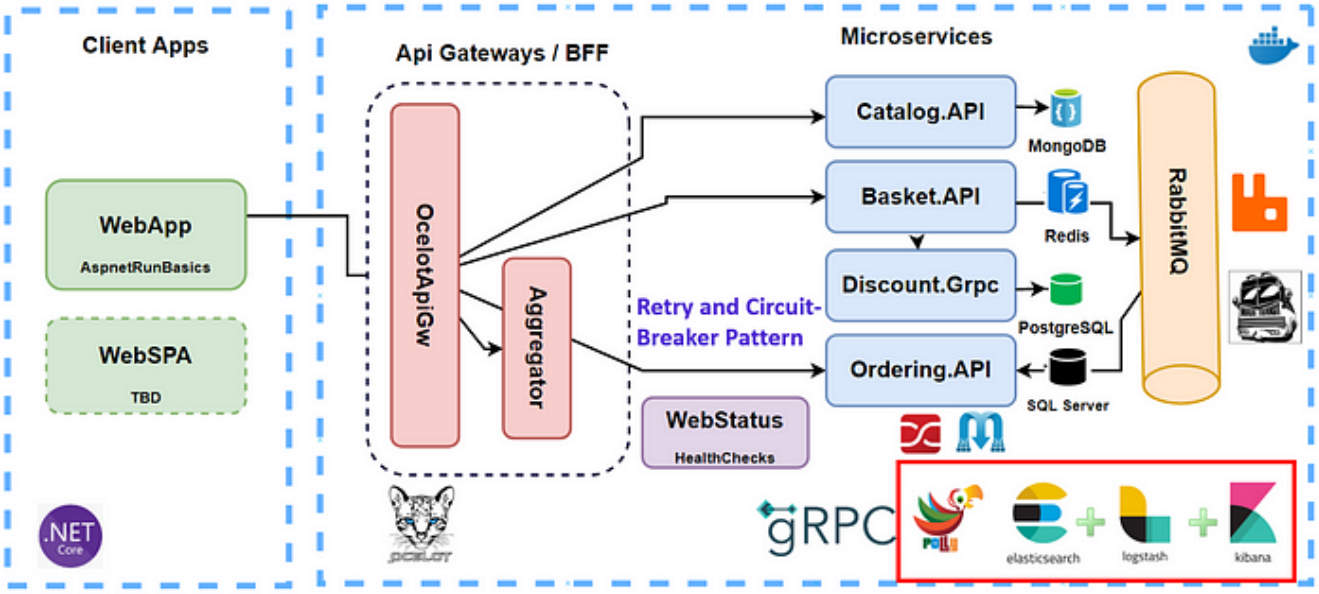
In this article we will show **how to perform Microservices Observability, Microservices Resilience and Monitoring** principles on .Net microservices.

So When you are developing projects in microservices architecture, it is crucial to following Microservices **Observability**, Microservices **Resilience** and **Monitoring** principles.

So, we will separate our Microservices Cross-Cutting Concerns in 4 main pillars;

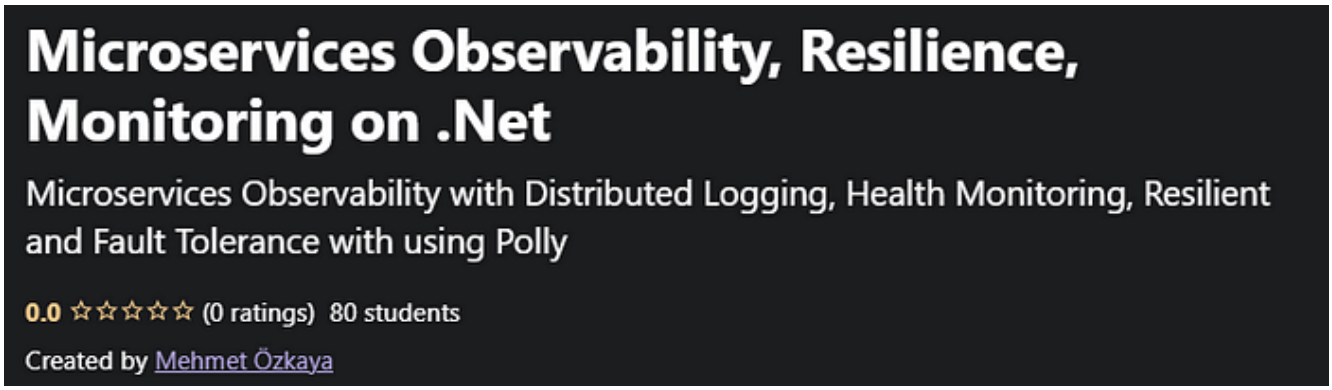
- Microservices **Observability** with Distributed Logging
- Microservices **Resilience** and **Fault Tolerance** with applying **Retry** and **Circuit-Breaker patterns** using **Polly**
- Microservices Monitoring with **Health Checks** using **WatchDog**
- Microservices Tracing with **OpenTelemetry** using **Zipkin**

So we are going to follow this 4 main pillars and develop our microservices reference application with using latest implementation and best practices on cloud-native microservices architecture.



See the our big picture what we are going to develop Microservices Architecture and Step by Step Implementation together.
We have already developed this microservices reference application in the microservices articles.

Step by Step Development w/ Udemy Course



Get Udemy Course with discounted — Microservices Observability, Resilience, Monitoring on .Net.

Source Code

Get the Source Code from AspnetRun Microservices Github — Clone or fork this repository, if you like don't forget the star. If you find or ask anything you can directly open issue on repository.

Microservices Cross-Cutting Concerns

So with this article, we will extend this microservices reference application with Cross-Cutting Concerns for provide microservices resilience.

We are going to cover Cross-Cutting Concerns in **4 main parts**;

1-Microservices Observability with Distributed Logging

This applying Elastic Stack which includes elasticsearh + logstach + kibana and SeriLog nuget package for .net microservices.
We will docker-compose Kibana image from docker hub and feed kibana with elastic stack.

2-Microservices Resilience and Fault Tolerance using Polly

This will apply retry and circuit-breaker patterns on microservices communication with creating polly policies.

3-Microservices Health Monitoring with using WatchDog

This will be the aspnet health check implementation with custom hc methods
which inludes database availabilities — for example in basket microservices, we will add sub-hc contitons for connecting Redis, and RabbitMQ.

4-Microservices Distributed Tracing with OpenTelemetry using Zipkin

This will be the implementation of OpenTelemetry with Zipkin.

By the end of this article, you'll learn how to design and developing Microservices Cross-Cutting Concerns — Microservices Observability with Distributed Logging, Health Monitoring, Resilient and Fault Tolerance with using Polly on .Net Microservices.

Background

This is the introduction of the series. This will be the series of articles. You can follow the series with below links.

Also you can find whole microservices series on below article;

Check for the previous article which explained overall microservice architecture of this repository.

We will focus on microservices cross-cutting concerns on these article series.

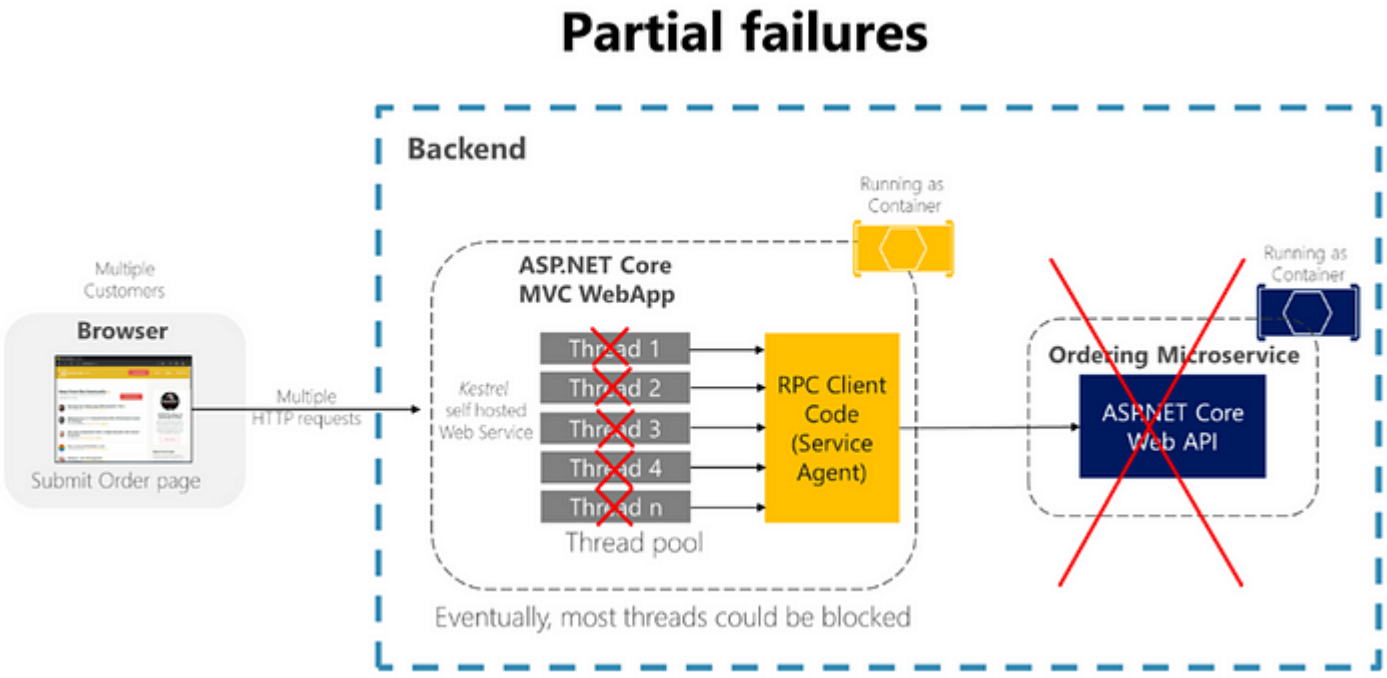
Prerequisites

- Install the .NET Core 5 or above SDK
- Install Visual Studio 2019 v16.x or above

- Docker Desktop

Microservices = Distributed Systems

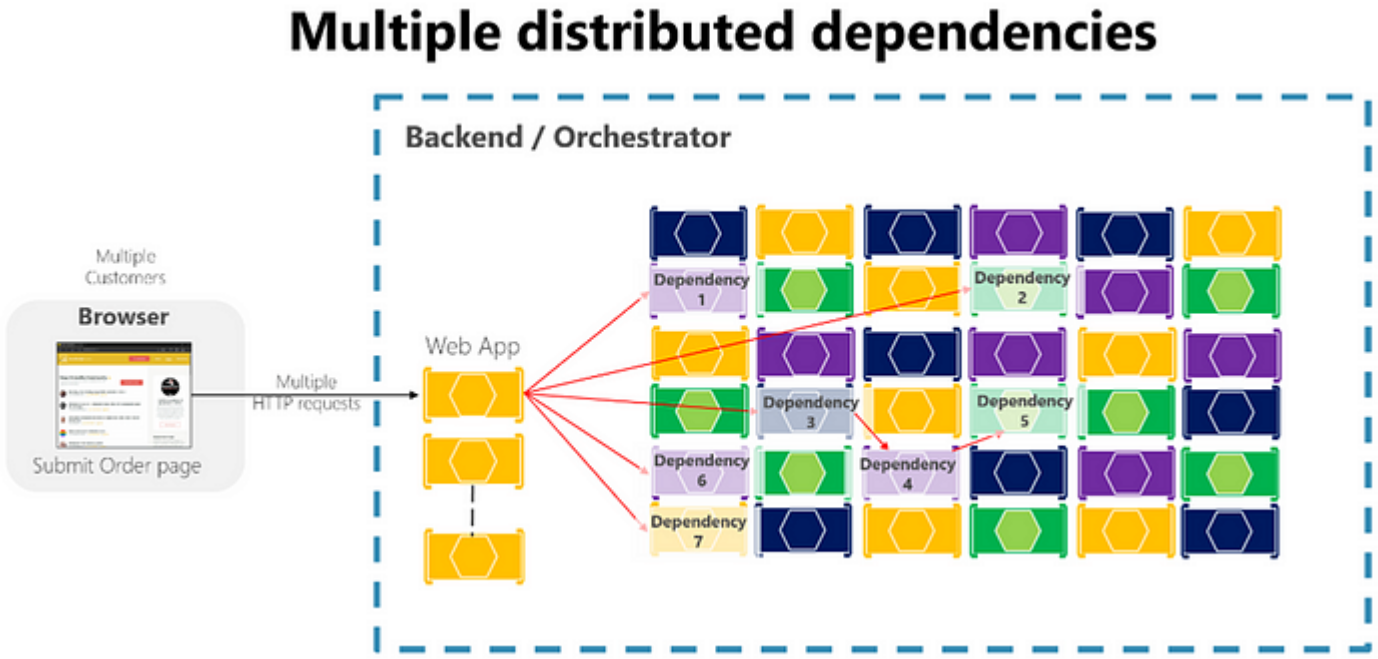
Microservice architecture have become the new model for building modern cloud-native applications. And microservices-based applications are distributed systems.



While architecting distributed microservices-based applications, it get lots of benefits like makes easy to scale and manage services, but as the same time, it is increasing interactions between those services have created a new set of problems.

Distributed Dependencies

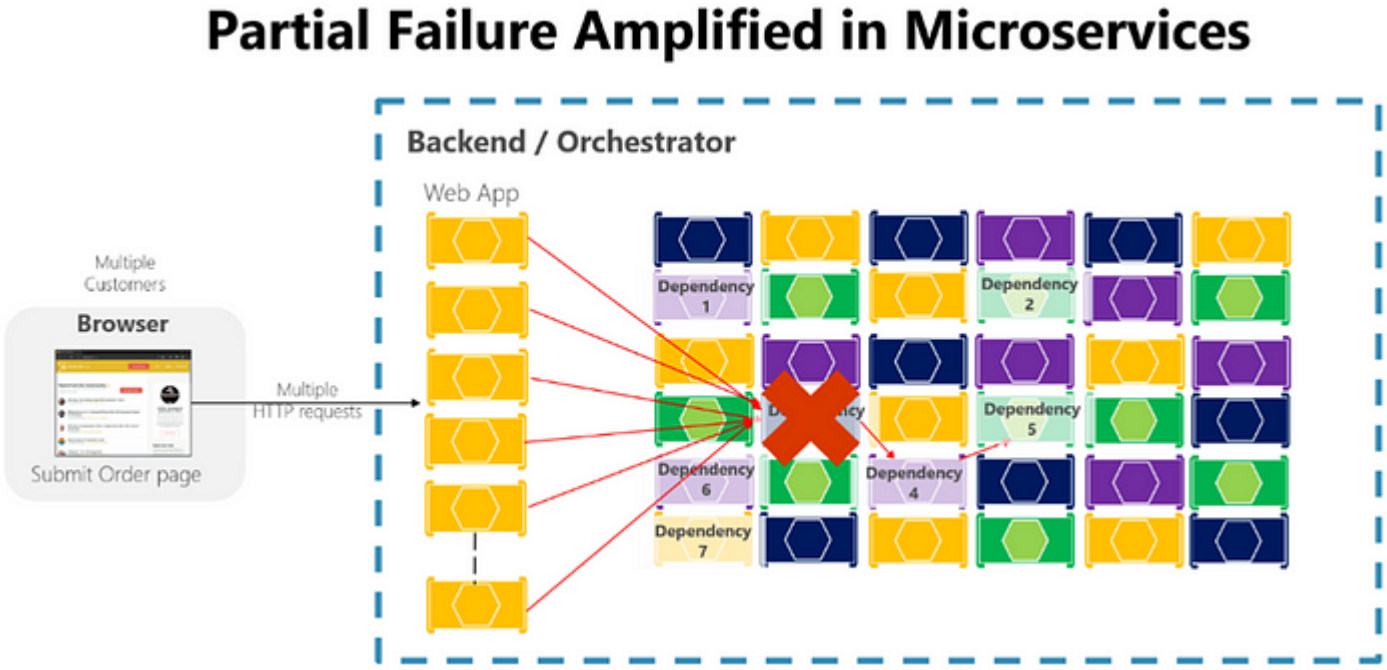
So we should assume that failures will happen, and we should dealing with unexpected failures, especially in a multiple Distributed Dependencies systems.



For example in case of network or container failures, microservices must have a strategy to retry requests again.

Dealing with Failures

But What happens when the machine where the microservice is running fails ?

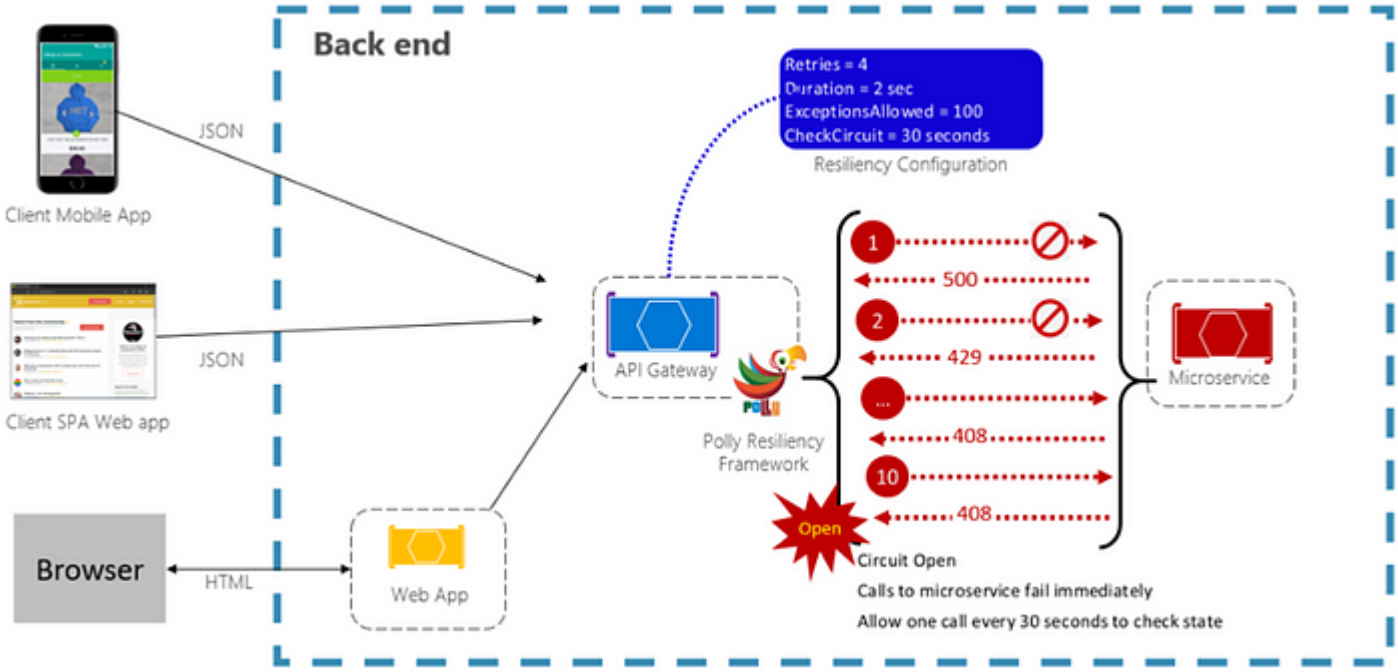


For instance, a single microservice can fail or might not be available to respond for a short time. Since clients and services are separate processes, a service might not be able to respond in a timely way to a client's request.
The service might be overloaded and responding very slowly to requests or might simply not be accessible for a short time because of network issues.

How do you handle the complexity that comes with distributed cloud-native microservices?

Microservices Resilience

Microservice should design for resiliency. A microservice needs to be resilient to failures and must accept partial failures. We should Design microservices to be resilient for these partial failures. microservices should ability to recover from failures and continue to function.

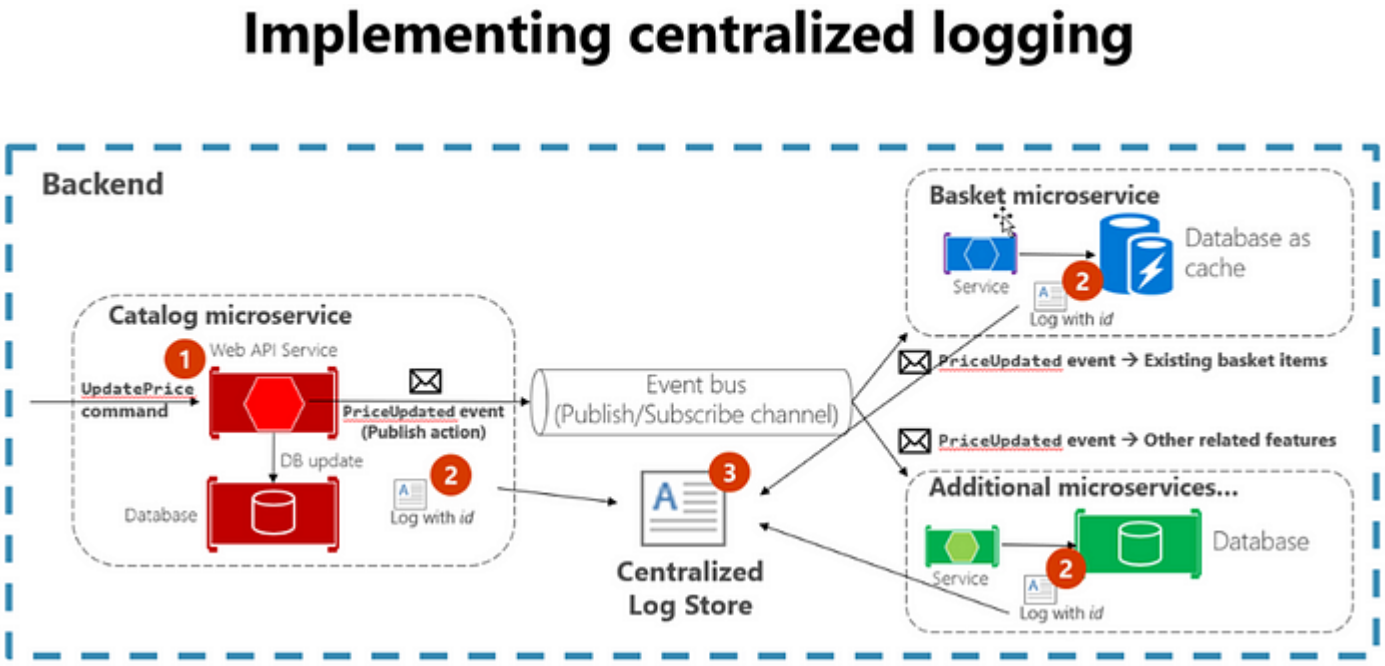


We should accepting failures and responding to them without any downtime or data loss. The main purpose of resiliency microservices is to return the application to a fully functioning state after a failure.

So when we are architecting distributed cloud applications, we should assume that failures will happen and design our microservices for resiliency. We accept that Microservices are going to fail at some point, thats why we need to learn embracing failures.

Microservices Observability

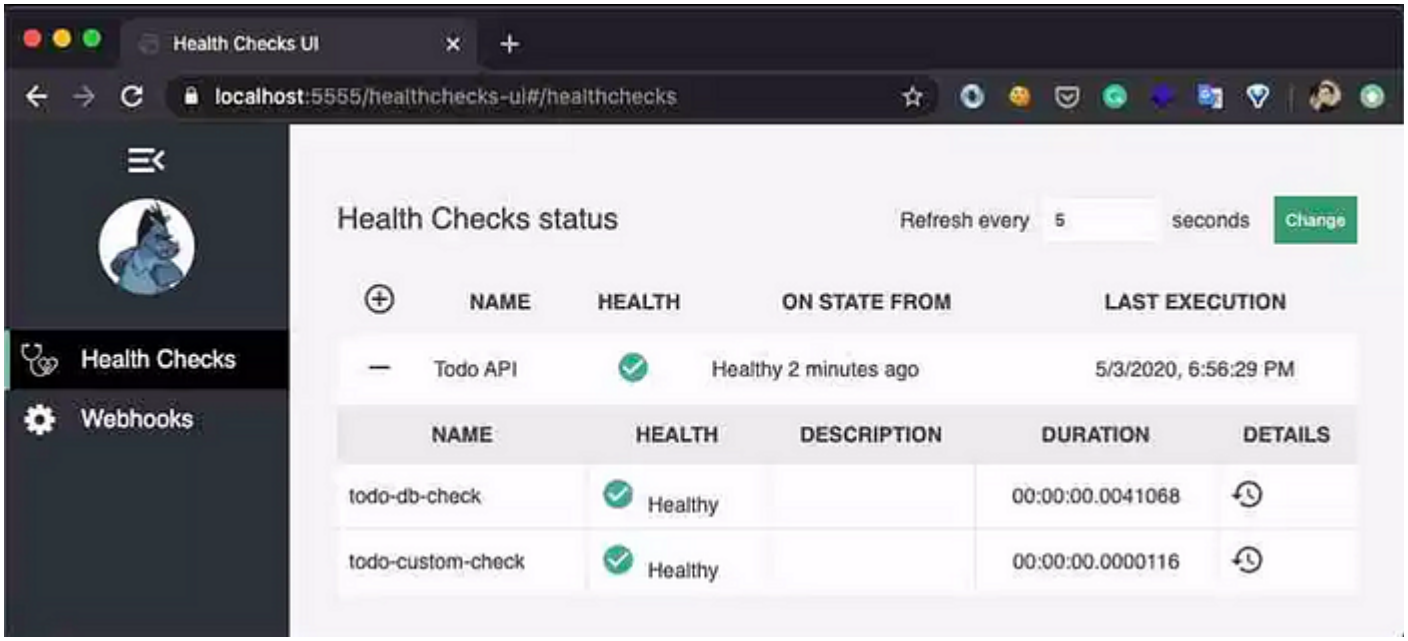
Also we should have a strategy for monitoring and managing the complex dependencies on microservices. That means we need to implement microservices observability with using distributed logging features.



Microservices Observability gives us greater operational insight and leading to understand incidents on our microservices architecture.

Microservices Monitoring

Also we should have microservices monitoring with health monitoring, Health monitoring is critical to multiple aspects of operating microservices. By this way, we can understand for a particular microservice is alive and ready to accommodate requests. We can also provide health information to our orchestrator's cluster, so that the cluster can act accordingly. For example Kubernetes has Liveness and Readiness probes that we can address health check urls.



That make a good health reporting which customized for our microservices like adding sub health checks for underlying database connection, and by this way we can detect and fix issues for our running application much more easily.

So, we will separate our Microservices Cross-Cutting Concerns in 4 main pillars;

- Microservices **Observability** with Distributed Logging
- Microservices **Resilience** and **Fault Tolerance** with applying **Retry** and **Circuit-Breaker patterns** using **Polly**
- Microservices Monitoring with **Health Checks** using **WatchDog**
- Microservices Tracing with **OpenTelemetry** using **Zipkin**