Proposed Microservices Architecture

esm_logo.jpg

Nexelus USA  
New York, NEW YORK (NY)  
646-558-1950 ext.128

**11/1/2023**

# Document Information

The following table shows the details for document creation, review, approval, and effective date.

| **Category** | **Information** |
| --- | --- |
| Work Product: | Proposed Microservices Architecture |
| Product Name: | Nexelus |
| Function Name: | SOC 1 Type II Document |
| Version: | 1.0 |
| Status: | Draft |
| Author(s): | Tauseef Shahzad |
| Reviewer(s): | Asim Jameel |
| Approver(s): | Imran Rahman |
| Control Status: | CONTROLLED, PROTECTED |
| Disclaimer: | This document contains confidential information. Do not distribute this document without prior approval from Nexelus. |

# Revision History

The following table is used for revision details of this document.

| **Author(s)** | **Date** | **Version** | **Description of Change** |
| --- | --- | --- | --- |
| Tauseef Shahzad | Nov 1, 2023 | 1.0 | Initial Draft |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Table of Contents

Contents

[Document Information ii](#_Toc149742291)

[Revision History iii](#_Toc149742292)

[Table of Contents i](#_Toc149742293)

[Scope 1](#_Toc149742294)

[Terms and Definitions 2](#_Toc149742295)

[Technologies 5](#_Toc149742296)

[Nexelus Micro Frontend 7](#_Toc149742297)

[Micro-Frontend Application - Inter-Communication 7](#_Toc149742298)

[State Management 7](#_Toc149742299)

[Global Store 8](#_Toc149742300)

[Nexelus Microservice Architecture 9](#_Toc149742301)

[API Gateway 9](#_Toc149742302)

[MicroService - Messaging 10](#_Toc149742303)

[Database Connectivity 11](#_Toc149742304)

[Authentication Service 11](#_Toc149742305)

[Cache 12](#_Toc149742306)

[Logging 12](#_Toc149742307)

[Multi-Tanent System 13](#_Toc149742308)

## Scope

Nexelus Application has been developed using Microsoft .Net Webforms. Some newer modules have been created using Microsoft .Net Core platform. .Net Core application is created as separate application, however both applications share the session/state information managed in Web Forms application.

The scope of this document is to outline Microservices architecture for Nexelus, and to propose high level architecture, technology, and application components. The document also covers proposed mechanism to integrate with existing application as well as how to implement Multi-Tenant Architecture.

## Terms and Definitions

At their cores, API gateways, Ingress controllers, and service meshes are each a type of proxy, designed to get traffic into and around your environments.

**Monolithic Applicaton Architecture**

A monolithic architecture is a traditional model of a software program, which is built as a unified unit that is self-contained and independent from other applications. A monolithic application may or may not consume external APIs and services.

**Microservice Application Architecture**

A microservices architecture is a type of application architecture where the application is developed as a collection of services. It provides the framework to develop, deploy, and maintain microservices architecture diagrams and services independently.

**Rest API**

REST API, or application programming interface, is a set of rules that define how applications or devices can connect to and communicate with each other. A REST API is an API that conforms to the design principles of the REST, or representational state transfer architectural style.

**Micro-Frontends**

Micro-frontend architecture is a design approach in which a front-end app is decomposed into individual, semi-independent “microapps” working loosely together. The micro-frontend concept is vaguely inspired by, and named after, microservices.

**API Gateway**

An API gateway routes API requests from a client to the appropriate services. But a big misunderstanding about this simple definition is the idea that an API gateway is a unique piece of technology. It’s not. Rather, “API gateway” describes a set of use cases that can be implemented via different types of proxies – most commonly an ADC or load balancer and reverse proxy, and increasingly an Ingress controller or service mesh.

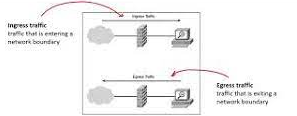
There isn’t a lot of agreement in the industry about what capabilities are “must haves” for a tool to serve as an API gateway. We typically see customers requiring the following abilities (grouped by use case):

* Resilience Use Cases
  + A/B testing, canary deployments, and blue‑green deployments
  + Protocol transformation (between JSON and XML, for example)
  + Rate limiting
  + Service discovery
* Traffic Management Use Cases
  + Method‑based routing and matching
  + Request/response header and body manipulation
  + Request routing at Layer 7
* Security Use Cases
  + API schema enforcement
  + Client authentication and authorization
  + Custom responses
  + Fine‑grained access control
  + TLS termination

Almost all these use cases are commonly used in Kubernetes. Protocol transformation and request/response header and body manipulation are less common since they’re generally tied to legacy APIs that aren’t well‑suited for Kubernetes and microservices environments.

**Ingress Controller**

An Ingress controller (also called a Kubernetes Ingress Controller – KIC for short) is a specialized Layer 4 and Layer 7 proxy that gets traffic into Kubernetes, to the services, and back out again (referred to as ingress‑egress or north‑south traffic). In addition to traffic management, Ingress controllers can also be used for visibility and troubleshooting, security, and identity, and all but the most advanced API gateway use cases.



**Service Mesh**

A service mesh handles traffic flowing between Kubernetes services (referred to as service-to-service or east‑west traffic) and is commonly used to achieve end-to-end encryption (E2EE). Service mesh adoption is small but growing as more organizations launch advanced deployments or have requirements for E2EE. A service mesh can be used as a distributed (lightweight) API gateway very close to the apps, made possible on the data plane level by service mesh sidecars.

**Container**

Containers are lightweight packages of your application code together with dependencies such as specific versions of programming language runtimes and libraries required to run your software services. Containers are packages of software that contain all the necessary elements to run in any environment. In this way, containers virtualize the operating system and run anywhere, from a private data center to the public cloud or even on a developer’s personal laptop.

**Docker**

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code, you can significantly reduce the delay between writing code and running it in production.

**Kubernetes**

Kubernetes automates operational tasks of container management and includes built-in commands for deploying applications, rolling out changes to your applications, scaling your applications up and down to fit changing needs, monitoring your applications, and more making it easier to manage applications.

**End-to-end encryption**

End-to-end encryption (E2EE) is a method of secure communication that prevents third parties from accessing data while it's transferred from one end system or device to another.

In E2EE, the data is encrypted on the sender's system or device, and only the intended recipient can decrypt it. As it travels to its destination, the message cannot be read or tampered with by an internet service provider (ISP), application service provider, hacker or any other entity or service.

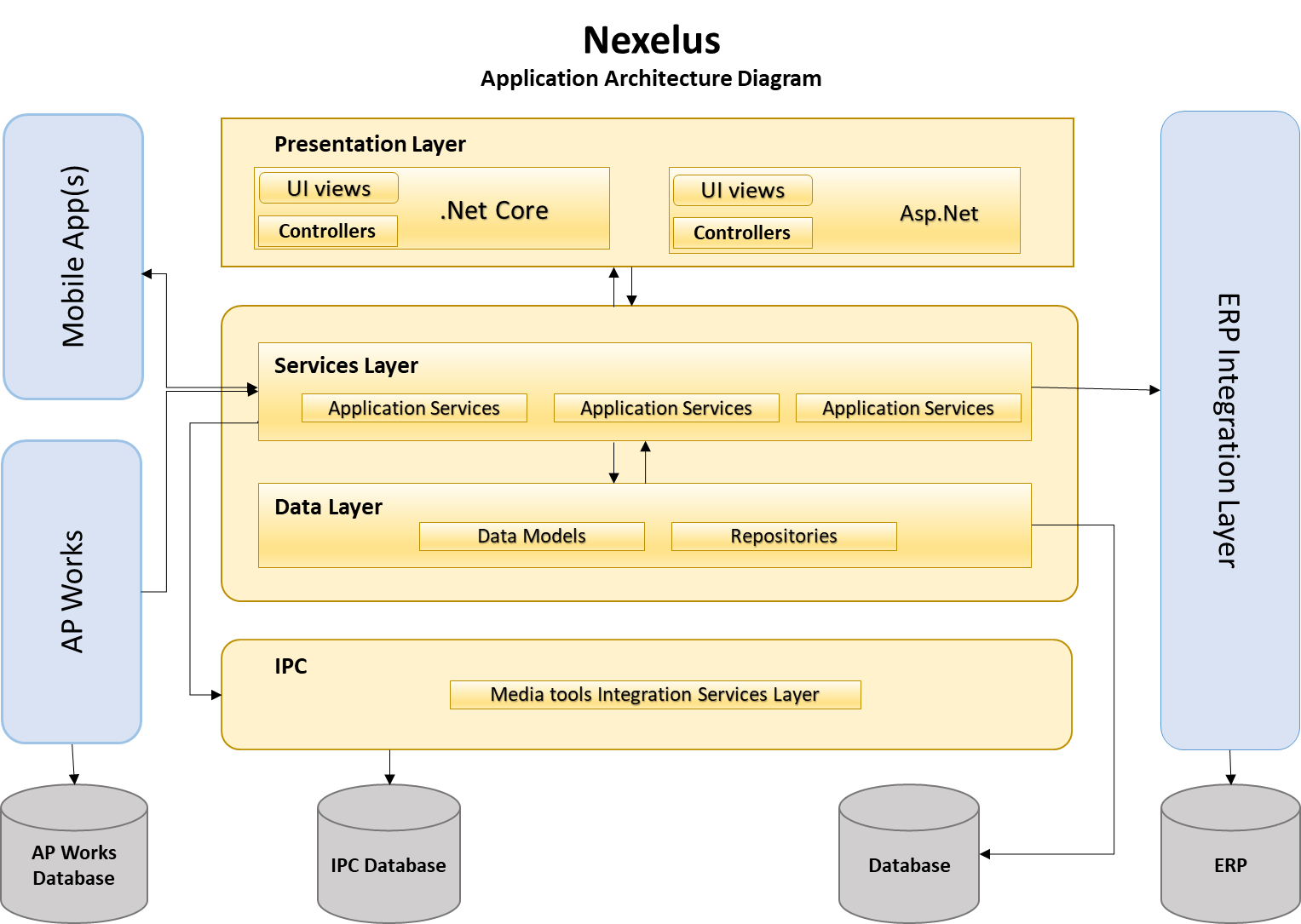
**Elasticsearch**

Elasticsearch is a search engine based on the Lucene library. It provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents.

## Technologies

#### Existing Architecture

Nexelus has been designed as dotNet Webforms Monolithic application. The application is divided in Presentation, Service and Database Layer. Some of its business functions are deployed as Rest services, such as IPC, etc. However, it is deployed as a Monolithic application that includes all business functions.



#### Proposed Architecture - (Micro Services with Micro Front Ends)

Microservices are a set of services that act together to make a whole application operate. This architecture utilizes APIs to pass information, such as user queries or a data stream, from one service to another.

How the underlying software works, or which hardware the service is built upon, depends solely on the team who built the service. This makes both communicating between teams and upgrading services very dynamic—even reactive—allowing a software company or team to be more resilient in its development.

React provides state-of-the-art functionality and is an excellent choice for developers looking for an easy-to-use and highly productive JavaScript framework. Using React, you can build complex UI interactions that communicate with the server in record time with JavaScript-driven pages.



## Micro Frontend Architecture

Micro frontends are an emerging architecture inspired by microservices. Building with micro frontends is a modern strategy for deconstructing a monolithic codebase into smaller parts, to increase the autonomy of each team and the delivery throughput.

Each part can be considered a micro frontend, which will be deployable independently of the others.

Nexelus application will be segregated between loosely coupled Micro Frontends and Microservices. The Micro Frontends will be developed primarily using Node and React. However other technologies may be included such as Vue or Angular as and when required basis in future.

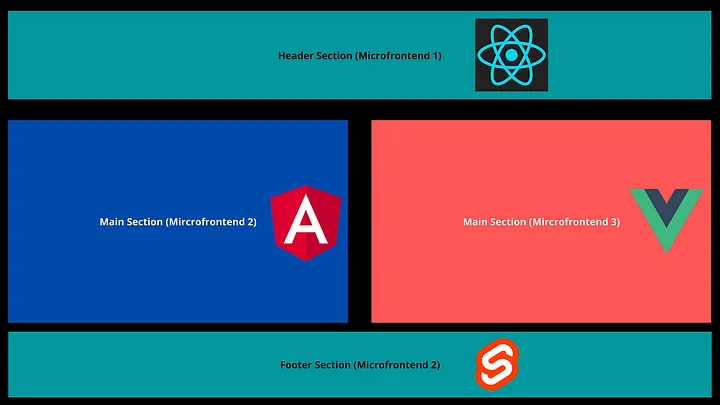
### Selecting Micro-Front Architecture

There are several popular approaches to implement Micro-front Architecture.

#### Micro-Federation

Module federation is a way to share and reuse modules across different webpack builds. It works by creating a central “entry” point for each module, which can then be imported and used by other modules. This allows for easy sharing of code and dependencies across teams, and makes it easy to update or replace individual modules without affecting the rest of the application. One of the main advantages of module federation is that it allows for a more modular and decoupled architecture. Each module can be developed and tested independently of the others, making it easier to make changes and fix bugs without affecting the rest of the application.

Another advantage of module federation is that it allows for better code reuse. By sharing code across modules, teams can avoid duplicating effort and reduce the overall size of the application. When using module federation with Angular, React, or Vue JS frameworks, it’s important to note that each module must be built using the same framework. This can be a limitation for teams that are using different frameworks within the same application.



#### Single-spa

Single-spa is a JavaScript library that allows for the creation of micro frontends within a single web application. It works by creating a “shell” application that loads and manages different micro frontends as they are needed. This allows for a more modular and decoupled architecture, similar to module federation. One of the main advantages of single-spa is that it allows for a more flexible architecture. Because each micro frontend is a standalone application, teams can use different frameworks and libraries within the same application. This can be especially useful for teams that are using different frameworks within the same application. Another advantage of single-spa is that it allows for better performance. By only loading the micro frontends that are needed at a given time, single-spa can reduce the overall size of the application and improve load times. When using single-spa with Angular, React, or Vue JS frameworks, it’s important to note that each micro frontend must be built as a standalone application. This can be more complex than using module federation, but it allows for more flexibility and better performance.

#### Proposed Solution

The main difference between module federation and single-spa is the way they handle the sharing and reuse of modules. Module federation uses a central “entry” point for each module, while single-spa uses a “shell” application to load and manage different micro frontends.

Another key difference is the flexibility of the architecture. Module federation requires that all modules be built using the same framework, while single-spa allows for the use of different frameworks within the same application.

In terms of performance, module federation can be more efficient when sharing code across modules, while single-spa can be more efficient when loading and managing micro frontends.

Finally, when using module federation with Angular, React, or Vue JS frameworks, it is important to note that each module must be built using the same framework, while single-spa allows for the use of different frameworks within the same application.

### Micro-Frontend Application - Inter-Communication

Micro frontend application comprises of multiple smaller apps running independently and embedded in a single Micro Frontends container application. The architecture with its whole list of advantages comes up with a set of complexities also and one of those is inter-communication of micro frontend applications. In this section, we are going to discuss five different techniques for micro frontend communications and the pros and cons of each one.

#### Props

This is the most basic technique for cross micro frontend communication where the container app is maintaining the state and passing it further to the required micro frontends.

* Pros
  + One of the very well-known techniques of data passing in component-based architecture.
  + Most of the frameworks support this way.
  + One can always use framework structs to avoid prop drilling issues e.g., React Context etc.
* Cons
  + Adds a lot of coupling between the micro frontends and the container app.
  + Hard to achieve if two micro frontends are not using the same framework.
  + It can impact the overall performance of the application as multiple unwanted layers will be re-rendered with every state change.

#### Platform Storage Apis

In this technique, we can leverage the platform's built-in storage APIs like Local Storage in browsers and Async Storage in cross-platform solutions like react native for mobile micro frontends also.

We can create a simple storage utility library exposing setter and getter from storage APIs. Now, instead of making micro frontends communicate via container app, each different micro frontend can set as well as read the data directly using the utility.

* Pros
  + Available for browsers as well as mobile devices. Local storage for browsers and Async storage for mobile apps.
  + Less coupling compares to passing props between the App and micro frontend but hard to debug which micro frontend is setting the data.
* Cons
  + Not a scalable solution for bigger applications. But can be used for a small set of data. It is always good to namespace the data set into platform storage according to the app name to avoid ambiguity.
  + Not a secure technique for saving protected data.

#### Custom Events

This technique is more suitable for web micro frontends and a more scalable technique for run-time micro frontends. The main idea here is to utilize the browser-inbuilt custom events APIs to publish the events with the data from one micro frontend and the other micro frontends subscribe to the events to get the data. This technique is closest to the events-driven architecture in the microservices world.

* Pros
  + The inbuilt solution in-browser platform.
  + Very much close to asynchronous event-based architecture in the microservices world. Easy for the backend developers to understand as well.
  + High setup cost but easy to scale.
  + Build a generic mechanism that all the micro frontend teams can follow.
* Cons
  + Not achievable in the case of mobile micro frontends

#### Custom Message bus

This technique is almost similar to the above one but instead of relying on the browser custom events API, we can build our pub-sub mechanism.

The message bus library can expose methods to publish, subscribe and unsubscribe the events. The publish event needs to make sure that all the subscribed handlers are invoked once the event is published.

* Pros
  + A custom-made solution equivalent to message bus in microservices implementation.
  + High setup cost but easy to scale.
  + Libraries like Postal.js (<https://github.com/postaljs/postal.js>) are available in the market.
* Cons
  + Similar to the custom events technique, it can be hard to make all the micro frontends teams follow the same pattern.

#### Post Message passing in iFrames

This technique is only relevant if you are building runtime micro frontends using iFrames. I feel that there is a better integration pattern for building micro frontends than using iFrames, but this technique can be very much useful when you need to inject a micro frontend inside another running app.

This technique is prone to hacking and brings up a lot of security concerns and hence it is not recommended.

#### Proposed Inter-Communication Technique:

It is proposed to use the **Custom Message Bus** for inter-application communication.

### State Management

Utilizing state management mechanisms in micro frontends is generally considered an unfavorable practice as it may result in the coupling of disparate applications, consequently undermining the benefits of adopting micro frontends. However, in certain scenarios where state sharing among micro frontends is unavoidable, it is recommended to keep the state as simple as possible to mitigate the potential drawbacks.

**Redux-Thunk Vs Redux-Saga**: Redux Thunk and Redux Saga take care of dealing with side effects. In very simple terms, applied to the most common scenario (async functions, specifically AJAX calls) Thunk allows Promises to deal with them, Saga uses Generators. Thunk is simple to use and Promises are familiar to many developers, Saga/Generators are more powerful but you will need to learn them. When Promises are just good enough, so is Thunk, when you deal with more complex cases on a regular basis, Saga gives you better tools.

As an example, what happens when you start an AJAX call in a route/view and then the user moves to a different one? Can you safely let the reducer change the state anyway? Saga makes it trivial to cancel the effect, Thunk requires you to take care of it, with solutions that don't scale as nicely.

// simple usage of redux-thunk:

export const asyncApiCall = values => {

return dispatch => {

return axios.get(url)

.then(response =>{

dispatch(successHandle(response));

})

.catch(error => {

dispatch(errorHandle(error));

});

};

};

// async/await usage of redux-thunk:

export const asyncApiCall = values => {

return async dispatch => {

try {

const response = await axios.get(url);

dispatch(successHandle(response));

}

catch(error) {

dispatch(errorHandle(error));

}

};

};

**Proposed Solution:**

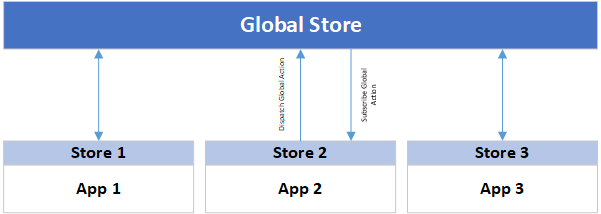
For managed, scalable, and controlled code, proposed solution is to use Redux-Saga.

### Global Store

The Global Store is not an actual store, rather it's a collection of multiple isolated Redux Stores. Each physical Redux Store here refers to the isolated store that each app uses. Micro frontends having access to the Global Store would be able to perform all operations that are allowed on an individual Redux Store including getState(), dispatch() and subscribe().

**Global Actions**

The concept of Global Action is available which allows other apps to dispatch actions to stores registered by other Micro Frontends. Each Micro Frontend has the capability to register a set of global actions along with the store. These sets of global actions can be dispatched in this Micro Frontend's store by other Micro Frontends. This enables cross-application communication.



**Cross-state callbacks**

Cross-application communication can also be achieved by subscribing to change notifications in other Micro Frontend's state. Since each micro-frontend has read-only permission to other states, they can also attach callbacks for listening to state changes. The callbacks can be attached either at an individual store level or at a global level (this would mean that state change in any store would invoke the callback).

### User Interface Libraries

#### Material-UI <https://mui.com/material-ui/>

[blurbs]

[More to be added]

## Nexelus Microservice Architecture

### API Gateway

The following API gateways are considered for Nexelus API Gateway.

**DAPR**<https://dapr.io/>

The Distributed Application Runtime (Dapr) provides APIs that simplify microservice connectivity. Whether your communication pattern is service to service invocation or pub/sub messaging, Dapr helps you write resilient and secured microservices. By letting Dapr’s sidecar take care of the complex challenges such as service discovery, message broker integration, encryption, observability, and secret management, you can focus on business logic and keep your code simple.

**Ocelot**<https://ocelot.readthedocs.io/>

A modern fast, scalable API gateway built on ASP.NET core. It is aimed at people using .NET running a micro service / service-oriented architecture that need a unified point of entry into their system. However, it will work with anything that speaks HTTP and run on any platform that ASP.NET Core supports. It manipulates the HttpRequest object into a state specified by its configuration until it reaches a request builder middleware where it creates a HttpRequestMessage object which is used to make a request to a downstream service.

**Proposed Architecture**:

API Gateway is a pivotal technology for Microservices application. Each of the above mentioned API Gateways has its own advantages.

Ocelot is an API gateway written in C# and is easiest to implement for dotNet Core applications.

DAPR is a Microsoft developed and backed API Gateway that is an actively developed and evolving platform.

ISTIO is most popular and has more deployments than the other two platforms.

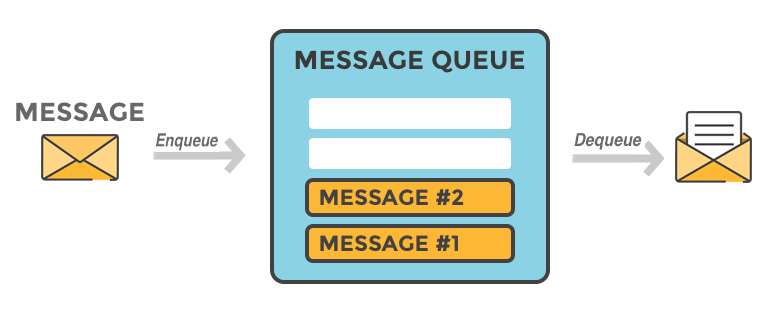
It is proposed to use Istio, however it is open for debate before we start actual development.

### MicroService - Messaging

We have multiple options to use for Messaging inside Microservice (Rabbit MQ, Azure Bus) ,

**RabbitMQ**<https://www.rabbitmq.com/>

RabbitMQ is a message-queueing software also known as a message broker or queue manager. Simply said; it is software where queues are defined, to which applications connect to transfer a message or messages.



**Azure Service Bus**<https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview>

Azure Service Bus is a fully managed enterprise message broker with message queues and publish-subscribe topics (in a namespace). Service Bus is used to decouple applications and services from each other, providing the following benefits:

* Load-balancing work across competing workers
* Safely routing and transferring data and control across service and application boundaries
* Coordinating transactional work that requires a high degree of reliability

**Saga design pattern**

<https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga>

The Saga design pattern is a way to manage data consistency across microservices in distributed transaction scenarios. A saga is a sequence of transactions that updates each service and publishes a message or event to trigger the next transaction step. If a step fails, the saga executes compensating transactions that counteract the preceding transactions.

A diagram of a service

Description automatically generated

**Proposed Messaging Architecture**

Need more R&D to propose messaging architecture.

### Database Connectivity

Nexelus Web Forms application uses ADO.net for database connectivity. Nexelus .Net core application is currently using Dapper as ORM.

An ORM or Object Relational Mapping is an object-oriented programming technique that is used to map data between two incompatible types. Dapper is called micro ORM as it does not provide the full functionality and features like Entity Framework does. Dapper provides you with an ability to fetch the data from a relational database. It does not support code first support approach and can work with the querying and update through raw SQL. It cannot be used to configure classes and has a set of extension methods for ADO.NET. Entity Framework is an open source ORM that is available as a part of .NET framework. It is responsible to handle the interactions  
between the relational databased and .NET applications. It simplifies the programming code by generating the model classes for the database schema automatically. It allows you to have the database first and create the entities from it or alternatively, the application code can be initiated before the designing of database. In this way, any manual changes to the database, needs to be updated or copied to the code level because the changes will be lost.

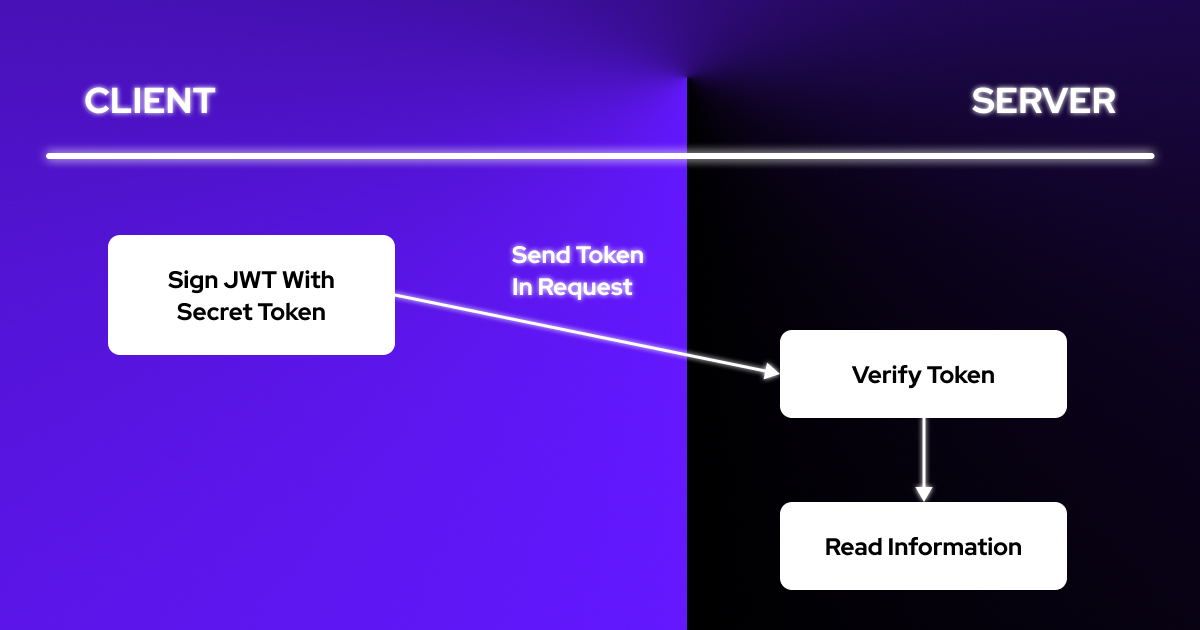
Reference: https://www.chubbydeveloper.com/entity-framework-vs-dapper/

### Authentication Service

* JWT
* OAuth OpenID
* Identity Server

[ Write a little details on each]

JWT is proposed for token base authentication.



JWT, or JSON Web Token, is a web protocol used to share security information between client and server. In a standard web application, private API requests contain JWT that is generated because of the verification of the user information, thus allowing these users to reach protected data and access services.

### Logging

This section covers logging of Microsoft Core Logging components.

##### log4net

The Apache log4net library is a tool to help the programmer output log statements to a variety of output targets. log4net is a port of the excellent Apache log4j™ framework to the Microsoft® .NET runtime, keeping the framework similar in spirit to the original log4j while taking advantage of new features in the .NET runtime. Over the years, development continued under the Apache Logging Services project.

##### NLog

NLog is legacy project that is used in existing Nexelus Application. Version 1.0 was released back in 2006, but it is still under active development with the latest version having been released in December of 2017. log4net hasn’t seen a release in 18 months, which isn’t necessarily bad as the project is stable. The latest release of NLog adds structured logging and support for .NET Standard.

##### Serilog

<https://stackify.com/nlog-vs-log4net-vs-serilog/>

The API is modern, it is easier to set up, it is better maintained, and it does structured logging by default. The ability to add enrichers gives you the ability to intercept and modify the messages is useful.

##### Grafana Loki <https://grafana.com/oss/loki/>

Loki is a horizontally scalable, highly available, multi-tenant log aggregation system inspired by Prometheus. It is designed to be very cost-effective and easy to operate. It does not index the contents of the logs, but rather a set of labels for each log stream.

**Proposed Logging Tool**: Logging is an important aspect to any application and shouldn’t be treated as an afterthought. Grafana Loki is more suitable for Docker applications as it allows viewing logs through portal using Kubernetes.

### Multi-Tanent System

We are going to have our Microservice to get requests from the Application and decide which DB (enterprise) going to use depending on the URL provided by the Frontend Application.

## Infrastructure and Monitoring

### Docker Desktop

It provides a straightforward GUI (Graphical User Interface) that lets you manage your containers, applications, and images directly from your machine. You can use Docker Desktop either on its own or as a complementary tool to the CLI.

Docker Desktop reduces the time spent on complex setups so you can focus on writing code. It takes care of port mappings, file system concerns, and other default settings, and is regularly updated with bug fixes and security updates.

### Kubernets

Kubernetes works by managing a cluster of compute instances and scheduling containers to run on the cluster based on the available compute resources and the resource requirements of each container. Containers are run in logical groupings called pods and you can run and scale one or many containers together as a pod.

Kubernetes control plane software decides when and where to run your pods, manages traffic routing, and scales your pods based on utilization or other metrics that you define. Kubernetes automatically starts pods on your cluster based on their resource requirements and automatically restarts pods if they or the instances they are running on fail. Each pod is given an IP address and a single DNS name, which Kubernetes uses to connect your services with each other and external traffic.

Kubernetes is required for local development and deployment of microservice infrastructure for development.

Kubernetes manages clusters of Amazon Elastic Compute Cloud (EC2) compute instances and runs containers on those instances with processes for deployment, maintenance, and scaling. Using Kubernetes, you can run any type of containerized applications using the same toolset on-premises and in the cloud.

### ISTIO

<https://istio.io/>

Istio is an open-source service mesh that layers transparently onto existing distributed applications. Istio’s powerful features provide a uniform and more efficient way to secure, connect, and monitor services. Istio is the path to load balancing, service-to-service authentication, and monitoring – with few or no service code changes. Its powerful control plane brings vital features, including:

* Secure service-to-service communication in a cluster with TLS encryption, strong identity-based authentication, and authorization
* Automatic load balancing for HTTP, gRPC, WebSocket, and TCP traffic
* Fine-grained control of traffic behavior with rich routing rules, retries, failovers, and fault injection.
* A pluggable policy layer and configuration API supporting access controls, rate limits and quotas.
* Automatic metrics, logs, and traces for all traffic within a cluster, including cluster ingress and egress.

### Prometheus

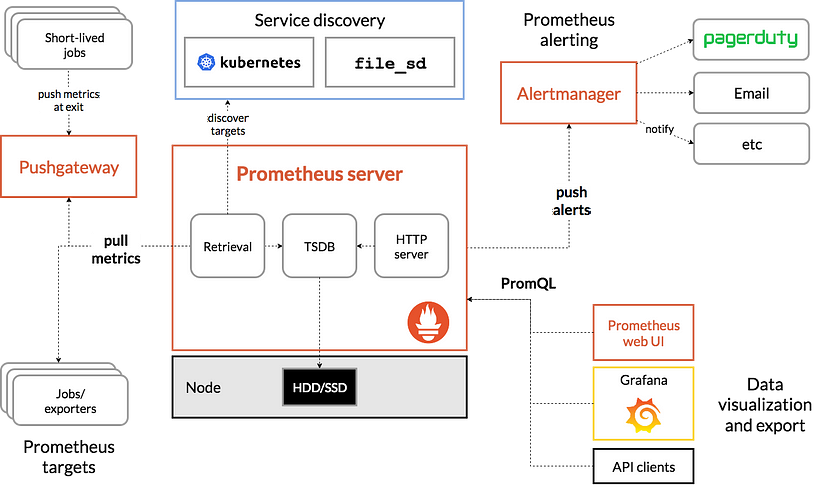
<https://prometheus.io/>

Prometheus is an open-source DevOps tool. It provides monitoring and real-time alerting functionality for container orchestration platforms like Kubernetes. It collects and stores the metrics from the platform as time series data. It has an out-of-box capability for monitoring the container orchestration platform. It acts as a data source for other data visualization libraries like Grafana.

The metrics that Prometheus collects from the Kubernetes cluster are:

* Kubernetes cluster health.
* CPU status.
* Memory usage.
* Kubernetes nodes status.
* Reports on potential performance bottlenecks.
* Performance metrics.
* Server resources.

The diagram below shows Prometheus’ components and how Prometheus works:



These are the components of Prometheus:

Prometheus Server is the core component in Prometheus architecture. It is where the actual monitoring job occurs.

Alertmanager alerts users via email and other communication channels, such as Slack.

Pushgateway can support temporary jobs. It allows users to push time series data to the Prometheus targets. It also handles metrics for short-lived jobs.

The prometheus server can further be split into three components:

* Data Retrieval Worker scraps and collects metrics data from the container orchestration platform. It then converts the metrics into time series data. It gathers metrics from many sources, specified in its setups.
* Time Series Database stores the time series data from the data retrieval component.
* HTTP Server responds to requests and PromQL queries for the time series data. It then presents the information in a web user interface or dashboard. It can either use a third-party platform like Grafana or the inbuilt Prometheus Web UI.

### Grafana

Grafana is a multi-platform, open-source online application for analytics and interactive visualization. When you connect it to supported data sources like Prometheus, it offers:

* Interactive Dashboards.
* Interactive charts.
* Graphs.
* Alerts for the web.

Grafana enables you to query, visualize and comprehend your metrics regardless of the data source. Besides Prometheus, Grafana supports several other data sources.

A screenshot of a computer

Description automatically generated

## Configure Docker, Kubernetes and Istio

A service mesh is a configurable infrastructure layer. It has capabilities to handle service-to-service communication, resilience, and many cross-cutting concerns. Proxy is a key component of service mesh. Sidecar proxy is injected in each service in this concept. Here I will show you service mesh communication using Istio with asp.net core applications on Kubernetes environment.

Istio is an open platform for providing a uniform way to integrate microservices, manage traffic flow across microservices, enforce policies and aggregate telemetry data. Istio uses following tools-

|  |  |
| --- | --- |
| Tool | Description |
| Prometheus | It monitors everything in the cluster. |
| Grafana | Data visualization tools |
| Jaeger | It’s used for distributed tracing. |

### Istio Configurations

Please follow the following steps to download, install, and configure Istio

|  |  |
| --- | --- |
| Step | Description |
| 1 | * Download Istio from following link:   <https://github.com/istio/istio/releases/tag/1.12.2>   * Extract zip file in a folder (e.g., d:\software\istio) * Add d:\software\istio\bin folder to Path   Add ISTIO\_HOME=”d:\software\istio” to windows environment variables. |
| 2 | * Verify Istio using following command   Istioctl |
| 3 | * Install Docker Desktop <https://docs.docker.com/desktop/install/windows-install/> |
| 4 | * Enable Kubernetes in Docker Desktop Settings:     It will install Kubernetes in Docker Desktop |
| 5 | * For installation, we use the demo configuration profile. It’s selected to have a good set of defaults for testing, but there are other profiles for production or performance testing. Use below command to install Istio.   istioctl install --set profile=demo -y |
| 6 | * Use the following command to verify Istio.   kubectl get all -n istio-system  You will see response similar to following |
| 7 | * Configure for auto proxy injection   Add a namespace label to instruct Istio to automatically inject Envoy sidecar proxies when you deploy your application later. Use below command to configure default namespance with Istio sidecar proxy.  kubectl label namespace default istio-injection=enabled |
| 8 | * Check label by using below command   kubectl describe namespace default |

## Microservice Application without Microfront

A Microservices sample application has been picked from the internet to explain implementation of Kubernetes, Istio, and ocelot application gateway. The code is placed on Teams in following location:

Location:

Explanation of individual projects and code files is given below:

### Create asp.net core applications

Create 4 asp.net core web api projects.

* Catalog.API,
* Location.API,
* Ordering.API
* BFF.Web.

### Catalog.API – API Project

Add the following nuget packages in the project.

* Install-Package Microsoft.EntityFrameworkCore.InMemory
* Install-Package Microsoft.EntityFrameworkCore.SqlServer
* Install-Package Microsoft.EntityFrameworkCore.Tools

Add a model class name Product in the model folder.

**Product.cs**

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace Catalog.API.Model

{

public class Product

{

[Key]

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public decimal Price { get; set; }

public int AvailableStock { get; set; }

public int RestockThreshold { get; set; }

}

}

Add CatalogContext class in Db folder.

CatalogContext.cs

using Catalog.API.Model;

using Microsoft.EntityFrameworkCore;

namespace Catalog.API.Db

{

public class CatalogContext : DbContext

{

public CatalogContext(DbContextOptions<CatalogContext> options) : base(options)

{

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

base.OnConfiguring(optionsBuilder);

}

public DbSet<Product> Products { get; set; }

}

}

Configure InMemory database and modify Program class as follows.

**Program.cs**

using Catalog.API.Db;

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

builder.Services.AddDbContext<CatalogContext>(opt => opt.UseInMemoryDatabase("CatalogDB"));

*// Learn more about configuring Swagger/OpenAPI at //*<https://aka.ms/aspnetcore/swashbuckle>

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

Create ProductsController in Controllers folder

**ProductsController.cs**

#nullable disable

using Catalog.API.Db;

using Catalog.API.Model;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

namespace Catalog.API.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class ProductsController : ControllerBase

{

private readonly CatalogContext \_context;

public ProductsController(CatalogContext context)

{

\_context = context;

}

// GET: api/Products

[HttpGet("GetAll")]

public async Task<ActionResult<IEnumerable<Product>>> GetProducts()

{

return await \_context.Products.ToListAsync();

}

// GET: api/Products/5

[HttpGet("{id}")]

public async Task<ActionResult<Product>> GetProduct(int id)

{

var product = await \_context.Products.FindAsync(id);

if (product == null)

{

return NotFound();

}

return product;

}

// PUT: api/Products/5

// To protect from overposting attacks, see <https://go.microsoft.com/fwlink/?linkid=2123754>

[HttpPut("Edit/{id}")]

public async Task<IActionResult> PutProduct(int id, Product product)

{

if (id != product.Id)

{

return BadRequest();

}

\_context.Entry(product).State = EntityState.Modified;

try

{

await \_context.SaveChangesAsync();

}

catch (DbUpdateConcurrencyException)

{

if (!ProductExists(id))

{

return NotFound();

}

else

{

throw;

}

}

return NoContent();

}

// POST: api/Products

*// To protect from overposting attacks, see*

*//* <https://go.microsoft.com/fwlink/?linkid=2123754>

[HttpPost("Add")]

public async Task<ActionResult<Product>> PostProduct(Product product)

{

\_context.Products.Add(product);

await \_context.SaveChangesAsync();

return CreatedAtAction("GetProduct", new { id = product.Id }, product);

}

// DELETE: api/Products/5

[HttpDelete("Delete/{id}")]

public async Task<IActionResult> DeleteProduct(int id)

{

var product = await \_context.Products.FindAsync(id);

if (product == null)

{

return NotFound();

}

\_context.Products.Remove(product);

await \_context.SaveChangesAsync();

return NoContent();

}

private bool ProductExists(int id)

{

return \_context.Products.Any(e => e.Id == id);

}

}

}

Add Dockerfile in the Catalog.API Project

**Dockerfile**

FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base

WORKDIR /app

EXPOSE 80

EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build

WORKDIR /src

COPY ["/Catalog.API.csproj", "Catalog.API/"]

RUN dotnet restore "Catalog.API/Catalog.API.csproj"

WORKDIR "/src/Catalog.API"

COPY . .

WORKDIR "/src/Catalog.API"

RUN dotnet build "Catalog.API.csproj" -c Release -o /app/build

FROM build AS publish

RUN dotnet publish "Catalog.API.csproj" -c Release -o /app/publish

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "Catalog.API.dll"]

Go to directory where dockerfile reside and run the following command to build docker image.

docker image build -t nexelus/catalog:1.0.1 .

Note: Don’t forgot to add . at the end of the command.

To configure pod add the following to file with code in Deploy/k8s folder

**deployment.yml**

# Configure Deployment

apiVersion: apps/v1

kind: Deployment

metadata:

name: catalogapi-deployment

spec:

selector:

matchLabels:

app: catalogapi-pod

template:

metadata:

labels:

app: catalogapi-pod

spec:

containers:

- name: catalogapi-container

image: nexelus/catalog:1.0.1

resources:

limits:

memory: "128Mi" # 128 mili bytes

cpu: "500m" # 500 mili cpu

ports:

- containerPort: 80

**service.yml**

# Configure service

apiVersion: v1

kind: Service

metadata:

name: catalogapi-service

spec:

selector:

app: catalogapi-pod

ports:

- port: 8001

targetPort: 80

type: LoadBalancer # *use LoadBalancer if you want to accesss out side of pod*

**Go to the Deploy/k8s directory and run the following commands.**

kubectl apply -f .\deployment.yml

kubectl apply -f .\service.yml

### Check pods have proxy auto-injected

By default istio will be injected automatically under this namespace.

Use the following command to check pods have proxy auto-injected.

kubectl get pods // To check pods

Output:

**NAME READY STATUS RESTARTS AGE**

catalogapi-deployment-68d56ccddd-sqfnj 2/2 Running 0 14m

Show the catalogapi proxy setup using the following command

kubectl describe pods catalogapi-deployment-68d56ccddd-sqfnj

Find all proxy container using the following command

docker container ls --filter name=istio-proxy\_\*

Check proxy processes for the catalogapi

docker container ls --filter name=istio-proxy\_catalogapi-deployment\* -q

### Location.API – API Project

Create a Controller name CountriesController in the Controllers folder as follows.

**CountriesController.cs**

using Microsoft.AspNetCore.Mvc;

namespace Location.API.Controllers

{

[ApiController]

[Route("api/[controller]")]

public class CountriesController : ControllerBase

{

[HttpGet("GetAll")]

public IEnumerable<string> Get()

{

return new string[] {"America","Bangladesh", "Canada" };

}

}

}

Add docker file in the project root directory as follows.

**Dockerfile**

FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base

WORKDIR /app

EXPOSE 80

EXPOSE 443

ENV ASPNETCORE\_URLS=http://\*:80;

ENV ASPNETCORE\_ENVIRONMENT=Development

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build

WORKDIR /src

COPY ["/Location.API.csproj", "Location.API/"]

RUN dotnet restore "Location.API/Location.API.csproj"

WORKDIR "/src/Location.API"

COPY . .

WORKDIR "/src/Location.API"

RUN dotnet build "Location.API.csproj" -c Release -o /app/build

FROM build AS publish

RUN dotnet publish "Location.API.csproj" -c Release -o /app/publish

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "Location.API.dll"]

Go to the root directory where Dockerfile reside and run the following command to build docker image.

docker image build -t nexelus/location:1.0.1 .

To configure pod add the following to file with code in Deploy/k8s folder

**deployment.yml**

apiVersion: apps/v1

kind: Deployment

metadata:

name: locationapi-deployment

spec:

selector:

matchLabels:

app: locationapi-pod

template:

metadata:

labels:

app: locationapi-pod

spec:

containers:

- name: locationapi-container

image: nexelus/location:1.0.1

resources:

limits:

memory: "128Mi" # 128 mili bytes

cpu: "500m" # 500 mili cpu

ports:

- containerPort: 80

**service.yml**

apiVersion: v1

kind: Service

metadata:

name: locationapi-service

spec:

selector:

app: locationapi-pod

ports:

- port: 8002

targetPort: 80

#type: LoadBalancer

Go to the Deploy/k8s directory and run the following commands.

kubectl apply -f .\deployment.yml

kubectl apply -f .\service.yml

### Ordering.API – API Project

Add the following nuget packages in the project.

Install-Package Microsoft.EntityFrameworkCore.InMemory

Install-Package Microsoft.EntityFrameworkCore.SqlServer

Install-Package Microsoft.EntityFrameworkCore.Tools

Create Order class in Models folders as follows.

**Order.cs**

namespace Ordering.API.Models

{

public class Order

{

public int Id { get; set; }

public string Address { get; set; }

public DateTime OrderDate { get; set; }

public string Comments { get; set; }

}

}

Create Ordering OrderingContext class in Db folder.

**OrderingContext.cs**

using Microsoft.EntityFrameworkCore;

using Ordering.API.Models;

namespace Ordering.API.Db

{

public class OrderingContext : DbContext

{

public OrderingContext(DbContextOptions<OrderingContext> options) : base(options)

{

}

public DbSet<Ordering.API.Models.Order> Order { get; set; }

}

}

Modify Program.cs to add dbcontext.

**Program.cs**

using Microsoft.EntityFrameworkCore;

using Ordering.API.Db;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

builder.Services.AddDbContext<OrderingContext>(opt => opt.UseInMemoryDatabase("CatalogDB"));

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

Create OrdersController in Controllers folder as follows.

**OrdersController.cs**

#nullable disable

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using Ordering.API.Db;

using Ordering.API.Models;

namespace Ordering.API.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class OrdersController : ControllerBase

{

private readonly OrderingContext \_context;

public OrdersController(OrderingContext context)

{

\_context = context;

}

// GET: api/Orders

[HttpGet("GetAll")]

public async Task<ActionResult<IEnumerable<Order>>> GetOrder()

{

return await \_context.Order.ToListAsync();

}

// GET: api/Orders/5

[HttpGet("{id}")]

public async Task<ActionResult<Order>> GetOrder(int id)

{

var order = await \_context.Order.FindAsync(id);

if (order == null)

{

return NotFound();

}

return order;

}

// PUT: api/Orders/5

// To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754

[HttpPut("Edit/{id}")]

public async Task<IActionResult> PutOrder(int id, Order order)

{

if (id != order.Id)

{

return BadRequest();

}

\_context.Entry(order).State = EntityState.Modified;

try

{

await \_context.SaveChangesAsync();

}

catch (DbUpdateConcurrencyException)

{

if (!OrderExists(id))

{

return NotFound();

}

else

{

throw;

}

}

return NoContent();

}

// POST: api/Orders

// To protect from overposting attacks, see

// https://go.microsoft.com/fwlink/?linkid=2123754

[HttpPost("Add")]

public async Task<ActionResult<Order>> PostOrder(Order order)

{

\_context.Order.Add(order);

await \_context.SaveChangesAsync();

return CreatedAtAction("GetOrder", new { id = order.Id }, order);

}

// DELETE: api/Orders/5

[HttpDelete("Delete/{id}")]

public async Task<IActionResult> DeleteOrder(int id)

{

var order = await \_context.Order.FindAsync(id);

if (order == null)

{

return NotFound();

}

\_context.Order.Remove(order);

await \_context.SaveChangesAsync();

return NoContent();

}

private bool OrderExists(int id)

{

return \_context.Order.Any(e => e.Id == id);

}

}

}

Create Docker file in the root directory

**Dockerfile**

FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base

WORKDIR /app

EXPOSE 80

EXPOSE 443

ENV ASPNETCORE\_URLS=http://\*:80;

ENV ASPNETCORE\_ENVIRONMENT=Development

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build

WORKDIR /src

COPY ["/Ordering.API.csproj", "Ordering.API/"]

RUN dotnet restore "Ordering.API/Ordering.API.csproj"

WORKDIR "/src/Ordering.API"

COPY . .

WORKDIR "/src/Ordering.API"

RUN dotnet build "Ordering.API.csproj" -c Release -o /app/build

FROM build AS publish

RUN dotnet publish "Ordering.API.csproj" -c Release -o /app/publish

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "Ordering.API.dll"]

Go to the directory where docker file exists and run the following command to build docker image.

docker image build -t nexelus/ordering:1.0.1 .

To configure pod add the following to file with code in Deploy/k8s folder

deployment.yml

apiVersion: apps/v1

kind: Deployment

metadata:

name: locationapi-deployment

spec:

selector:

matchLabels:

app: locationapi-pod

template:

metadata:

labels:

app: locationapi-pod

spec:

containers:

- name: locationapi-container

image: nexelus/location:1.0.1

resources:

limits:

memory: "128Mi" # 128 mili bytes

cpu: "500m" # 500 mili cpu

ports:

- containerPort: 80

service.yml

apiVersion: v1

kind: Service

metadata:

name: orderingapi-service

spec:

selector:

app: orderingapi-pod

ports:

- port: 8003

targetPort: 80

type: LoadBalancer

Go to the Deploy/k8s directory and run the following commands.

kubectl apply -f .\deployment.yml

kubectl apply -f .\service.yml

### BFF.Web - API Gateway Project

Install the following nuget packages in the project.

Install-Package Ocelot

Install-Package Ocelot.Cache.CacheManager

Install-Package MMLib.SwaggerForOcelot

Install-Package Ocelot.Provider.Polly

Create a folder name Routes/Routes.dev and add the following files in that folder

**ocelot.catalog.api.json**

{

"Routes": [

{

"DownstreamPathTemplate": "/{everything}",

"DownstreamScheme": "https",

"SwaggerKey": "catalog",

"DownstreamHostAndPorts": [

{

"Host": "localhost",

"Port": "7282"

}

],

"UpstreamPathTemplate": "/catalog/{everything}",

"UpstreamHttpMethod": [

"GET",

"POST",

"PUT",

"DELETE"

]

}

]

}

**ocelot.location.api.json**

{

"Routes": [

{

"DownstreamPathTemplate": "/{everything}",

"DownstreamScheme": "https",

"SwaggerKey": "location",

"DownstreamHostAndPorts": [

{

"Host": "localhost",

"Port": "7003"

}

],

"UpstreamPathTemplate": "/location/{everything}",

"UpstreamHttpMethod": [

"GET",

"POST",

"PUT",

"DELETE"

]

}

]

}

**ocelot.ordering.api.json**

{

"Routes": [

{

"DownstreamPathTemplate": "/{everything}",

"DownstreamScheme": "https",

"SwaggerKey": "ordering",

"DownstreamHostAndPorts": [

{

"Host": "localhost",

"Port": "7126"

}

],

"UpstreamPathTemplate": "/ordering/{everything}",

"UpstreamHttpMethod": [

"GET",

"POST",

"PUT",

"DELETE"

]

}

]

}

**ocelot.global.json**

{

"GlobalConfiguration": {

"BaseUrl": "http://localhost:5205"

}

}

ocelot.SwaggerEndPoints.json

{

"SwaggerEndPoints": [

{

"Key": "bffweb",

"TransformByOcelotConfig": false,

"Config": [

{

"Name": "BFF.Web",

"Version": "1.0",

"Url": "http://localhost:5205/swagger/v1/swagger.json"

}

]

},

{

"Key": "location",

"TransformByOcelotConfig": true,

"Config": [

{

"Name": "Location.API",

"Version": "1.0",

"Url": "http://localhost:5205/location/swagger/v1/swagger.json"

}

]

},

{

"Key": "catalog",

"TransformByOcelotConfig": true,

"Config": [

{

"Name": "Catalog.API",

"Version": "1.0",

"Url": "http://localhost:5205/catalog/swagger/v1/swagger.json"

}

]

},

{

"Key": "ordering",

"TransformByOcelotConfig": true,

"Config": [

{

"Name": "Ordering.API",

"Version": "1.0",

"Url": "http://localhost:5205/ordering/swagger/v1/swagger.json"

}

]

}

]

}

Create a folder name Routes/Routes.prod and add the following files in that folder

**ocelot.catalog.api.json**

{

"Routes": [

{

"DownstreamPathTemplate": "/{everything}",

"DownstreamScheme": "http",

"SwaggerKey": "catalog",

"DownstreamHostAndPorts": [

{

"Host": "catalogapi-service",

"Port": "8001"

}

],

"UpstreamPathTemplate": "/catalog/{everything}",

"UpstreamHttpMethod": [

"GET",

"POST",

"PUT",

"DELETE"

]

}

]

}

**ocelot.location.api.json**

{

"Routes": [

{

"DownstreamPathTemplate": "/{everything}",

"DownstreamScheme": "http",

"SwaggerKey": "location",

"DownstreamHostAndPorts": [

{

"Host": "locationapi-service",

"Port": "8002"

}

],

"UpstreamPathTemplate": "/location/{everything}",

"UpstreamHttpMethod": [

"GET",

"POST",

"PUT",

"DELETE"

]

}

]

}

**ocelot.ordering.api.json**

{

"Routes": [

{

"DownstreamPathTemplate": "/{everything}",

"DownstreamScheme": "http",

"SwaggerKey": "ordering",

"DownstreamHostAndPorts": [

{

"Host": "orderingapi-service",

"Port": "8003"

}

],

"UpstreamPathTemplate": "/ordering/{everything}",

"UpstreamHttpMethod": [

"GET",

"POST",

"PUT",

"DELETE"

]

}

]

}

**ocelot.global.json**

{

"GlobalConfiguration": {

"BaseUrl": "http://bffweb-service:8011"

}

}

**ocelot.SwaggerEndPoints.json**

{

"SwaggerEndPoints": [

{

"Key": "bffweb",

"TransformByOcelotConfig": false,

"Config": [

{

"Name": "BFF.Web",

"Version": "1.0",

"Url": "http://bffweb-service:8011/swagger/v1/swagger.json"

}

]

},

{

"Key": "location",

"TransformByOcelotConfig": true,

"Config": [

{

"Name": "Location.API",

"Version": "1.0",

"Url": "http://bffweb-service:8011/location/swagger/v1/swagger.json"

}

]

},

{

"Key": "catalog",

"TransformByOcelotConfig": true,

"Config": [

{

"Name": "Catalog.API",

"Version": "1.0",

"Url": "http://bffweb-service:8011/catalog/swagger/v1/swagger.json"

}

]

},

{

"Key": "ordering",

"TransformByOcelotConfig": true,

"Config": [

{

"Name": "Ordering.API",

"Version": "1.0",

"Url": "http://bffweb-service:8011/catalog/swagger/v1/swagger.json"

}

]

}

]

}

Add AlterUpstream Class in Config Folder

**AlterUpstream.cs**

using Newtonsoft.Json;

using Newtonsoft.Json.Linq;

namespace BFF.Web.Config

{

public class AlterUpstream

{

public static string AlterUpstreamSwaggerJson(HttpContext context, string swaggerJson)

{

var swagger = JObject.Parse(swaggerJson);

// ... alter upstream json

return swagger.ToString(Formatting.Indented);

}

}

}

Modify Program.cs to configure ocelot

**Program.cs**

using Catalog.API.Db;

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

builder.Services.AddDbContext<CatalogContext>(opt => opt.UseInMemoryDatabase("CatalogDB"));

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

Add docker file in the root directory.

**Dockerfile**

#See https://aka.ms/containerfastmode to understand how Visual Studio uses this Dockerfile to build your images for faster debugging.

FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base

WORKDIR /app

EXPOSE 80

EXPOSE 443

ENV ASPNETCORE\_URLS=http://\*:80;

ENV ASPNETCORE\_ENVIRONMENT=Development

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build

WORKDIR /src

COPY ["/BFF.Web.csproj", "BFF.Web/"]

RUN dotnet restore "BFF.Web/BFF.Web.csproj"

WORKDIR "/src/BFF.Web"

COPY . .

WORKDIR "/src/BFF.Web"

RUN dotnet build "BFF.Web.csproj" -c Release -o /app/build

FROM build AS publish

RUN dotnet publish "BFF.Web.csproj" -c Release -o /app/publish

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "BFF.Web.dll"]

Go to director where dockerfile reside and run the following command to build docker image.

docker image build -t nexelus/bff.web:1.0.1 .

To configure pod and service add the following yml file with code in Deploy/k8s folder

**deployment.yml**

apiVersion: apps/v1

kind: Deployment

metadata:

name: bffweb-deployment

spec:

selector:

matchLabels:

app: bffweb-pod

template:

metadata:

labels:

app: bffweb-pod

spec:

containers:

- name: bffweb-container

image: nexelus/bff.web:1.0.1

resources:

limits:

memory: "128Mi"

cpu: "500m"

ports:

- containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

name: bffweb-service

spec:

selector:

app: bffweb-pod

ports:

- port: 8011

targetPort: 80

type: LoadBalancer

Go to the Deploy/k8s directory and run the following commands.

kubectl apply -f .\deployment.yml

Now you can run services using in the kubernetes using the following command

kubectl get svc

**Output:**

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

bffweb-service LoadBalancer 10.111.186.235 localhost 8011:31690/TCP 10m

catalogapi-service LoadBalancer 10.101.130.94 localhost 8001:30710/TCP 138m

kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 3d20h

locationapi-service ClusterIP 10.100.204.33 <none> 8002/TCP 7m57s

orderingapi-service LoadBalancer 10.96.12.11 localhost 8003:31264/TCP 15m

You can now access catalog api, location api and ordering api using bffweb’s swagger defination

Check all end point using api gateway and swagger using the following URL

http://localhost:8011/swagger/index.html

Select Swagger definition from top right corner of BFF

### Visualize Service Mesh

**Install Kiali dashboard**

Go to the Istio’s directory. Install Kiali and other addons and wait for them to be deployed. Execute below command inside Istio folder. Use Git bash instead of powershell.

kubectl apply -f samples/addons

Execute below command and wait till get success roll out message.

kubectl rollout status deployment/kiali -n istio-system

Note: If there are errors trying to install the addons, try running the command again. There may be some timing issues which will be resolved when the command is run again.

Verify the deployment with below command.

kubectl get po -n istio-system

**Output:**

NAME READY STATUS RESTARTS AGE

grafana-6ccd56f4b6-sc894 1/1 Running 0 13m

istio-egressgateway-c9cbbd99f-wk265 1/1 Running 0 87m

istio-ingressgateway-7c8bc47b49-xpvvc 1/1 Running 0 87m

istiod-765596f7ff-2p72v 1/1 Running 0 89m

jaeger-5d44bc5c5d-g2wcl 1/1 Running 0 13m

kiali-79b86ff5bc-cqwrp 1/1 Running 0 13m

prometheus-64fd8ccd65-lglld 2/2 Running 0 13m

Now run the Kiali dashboard using the below command

istioctl dashboard kiali

Kiali dashboard will be open.

Hit the gateway URL. Use the following URL and hit several times and you will get the reflect in kiali dashbaord as below.

http://localhost:8011/swagger/index.html

Select Swagger definition from top right corner of BFF

A screenshot of a computer

Description automatically generated

Step 12: Monitoring with Prometheus & Grafana

Check Prometheus and Grafana is running using the following command.

kubectl get po -n istio-system

Run Prometheus dashboard using the following command

istioctl dashboard prometheus

View graph in diffrent ways like -

* Select istio\_requests\_total.
* Switch to Graph.
* Check Status/Targets - Kubernetes service discovery.

A screenshot of a computer

Description automatically generated

Run Grafana dashboard using the following command

istioctl dashboard grafana

Go to Dashboar->Manage->Istio and see the dashboar as below.

A screenshot of a computer

Description automatically generated

### Jaegar UI - Distributed Tracing

Run Jaeger UI using the following command

istioctl dashboard jaeger

A screenshot of a computer

Description automatically generated

### Step 14: Logging from Istio and Envoy

Create a YAML file and name the file elasticsearch.yaml and write below code.

**elasticsearch.yaml**

# Logging Namespace. All below are a part of this namespace.

apiVersion: v1

kind: Namespace

metadata:

name: logging

---

# Elasticsearch Service

apiVersion: v1

kind: Service

metadata:

name: elasticsearch

namespace: logging

labels:

app: elasticsearch

spec:

ports:

- port: 9200

protocol: TCP

targetPort: db

selector:

app: elasticsearch

---

# Elasticsearch Deployment

apiVersion: apps/v1

kind: Deployment

metadata:

name: elasticsearch

namespace: logging

labels:

app: elasticsearch

spec:

replicas: 1

selector:

matchLabels:

app: elasticsearch

template:

metadata:

labels:

app: elasticsearch

annotations:

sidecar.istio.io/inject: "false"

spec:

containers:

- image: docker.elastic.co/elasticsearch/elasticsearch-oss:6.1.1

name: elasticsearch

resources:

# need more cpu upon initialization, therefore burstable class

limits:

cpu: 1000m

requests:

cpu: 100m

env:

- name: discovery.type

value: single-node

ports:

- containerPort: 9200

name: db

protocol: TCP

- containerPort: 9300

name: transport

protocol: TCP

volumeMounts:

- name: elasticsearch

mountPath: /data

volumes:

- name: elasticsearch

emptyDir: {}

Create a YAML file and name the file kibana.yaml and write below code.

**kibana.yaml**

# Kibana Service

apiVersion: v1

kind: Service

metadata:

name: kibana

namespace: logging

labels:

app: kibana

spec:

ports:

- port: 5601

protocol: TCP

targetPort: ui

selector:

app: kibana

---

# Kibana Deployment

apiVersion: apps/v1

kind: Deployment

metadata:

name: kibana

namespace: logging

labels:

app: kibana

spec:

replicas: 1

selector:

matchLabels:

app: kibana

template:

metadata:

labels:

app: kibana

annotations:

sidecar.istio.io/inject: "false"

spec:

containers:

- name: kibana

image: docker.elastic.co/kibana/kibana-oss:6.1.1

resources:

# need more cpu upon initialization, therefore burstable class

limits:

cpu: 1000m

requests:

cpu: 100m

env:

- name: ELASTICSEARCH\_URL

value: http://elasticsearch:9200

ports:

- containerPort: 5601

name: ui

protocol: TCP

---

apiVersion: networking.istio.io/v1alpha3

kind: Gateway

metadata:

name: kibana-gateway

namespace: logging

spec:

selector:

istio: ingressgateway

servers:

- port:

number: 15033

name: http-kibana

protocol: HTTP

hosts:

- "\*"

---

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

name: kibana-vs

namespace: logging

spec:

hosts:

- "\*"

gateways:

- kibana-gateway

http:

- match:

- port: 15033

route:

- destination:

host: kibana

port:

number: 5601

Create a YAML file and name the file fluentd.yaml and write below code.

**fluentd.yaml**

apiVersion: v1

kind: ServiceAccount

metadata:

name: fluentd

namespace: kube-system

---

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: fluentd

namespace: kube-system

rules:

- apiGroups:

- ""

resources:

- pods

- namespaces

verbs:

- get

- list

- watch

---

kind: ClusterRoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: fluentd

roleRef:

kind: ClusterRole

name: fluentd

apiGroup: rbac.authorization.k8s.io

subjects:

- kind: ServiceAccount

name: fluentd

namespace: kube-system

---

# Fluentd Service

apiVersion: v1

kind: Service

metadata:

name: fluentd-es

namespace: kube-system

labels:

app: fluentd-es

spec:

ports:

- name: fluentd-tcp

port: 24224

protocol: TCP

targetPort: 24224

- name: fluentd-udp

port: 24224

protocol: UDP

targetPort: 24224

selector:

k8s-app: fluentd-logging

---

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: fluentd

namespace: kube-system

labels:

k8s-app: fluentd-logging

version: v1

kubernetes.io/cluster-service: "true"

spec:

selector:

matchLabels:

k8s-app: fluentd-logging

template:

metadata:

labels:

k8s-app: fluentd-logging

version: v1

kubernetes.io/cluster-service: "true"

spec:

serviceAccount: fluentd

serviceAccountName: fluentd

tolerations:

- key: node-role.kubernetes.io/master

effect: NoSchedule

containers:

- name: fluentd

image: fluent/fluentd-kubernetes-daemonset:v1.3-debian-elasticsearch

env:

- name: FLUENT\_ELASTICSEARCH\_HOST

value: "elasticsearch.logging"

- name: FLUENT\_ELASTICSEARCH\_PORT

value: "9200"

- name: FLUENT\_ELASTICSEARCH\_SCHEME

value: "http"

- name: FLUENT\_UID

value: "0"

resources:

limits:

memory: 200Mi

requests:

cpu: 100m

memory: 200Mi

volumeMounts:

- name: varlog

mountPath: /var/log

- name: varlibdockercontainers

mountPath: /var/lib/docker/containers

readOnly: true

terminationGracePeriodSeconds: 30

volumes:

- name: varlog

hostPath:

path: /var/log

- name: varlibdockercontainers

hostPath:

path: /var/lib/docker/containers

Now execute all above files with the commands below.

kubectl apply -f elasticsearch.yaml

kubectl apply -f kibana.yaml

kubectl apply -f fluentd.yaml

kubectl get pods -n logging

If you are using docker desktop you can use below command to port forward.

kubectl port-forward svc/kibana 8099:5601 -n logging

Now browse kibana using http://localhost:8099/

A screenshot of a computer

Description automatically generated

### Configure Istio to Log to Fluentd

Now we are going to configure Istio to use the same FluentD instance, and send proxy logs through FluentD into Elasticsearch. It will be actual adapter configuration that I mentioned earler.

Create a YAMl file and name the file fluentd-istio.yaml and write below code.

fluentd-istio.yaml

# Configuration for logentry instances

apiVersion: config.istio.io/v1alpha2

kind: instance

metadata:

name: newlog

namespace: istio-system

spec:

compiledTemplate: logentry

params:

severity: '"info"'

timestamp: request.time

variables:

source: source.labels["app"] | source.workload.name | "unknown"

user: source.user | "unknown"

destination: destination.labels["app"] | destination.workload.name | "unknown"

responseCode: response.code | 0

responseSize: response.size | 0

latency: response.duration | "0ms"

monitored\_resource\_type: '"UNSPECIFIED"'

---

# Configuration for a Fluentd handler

apiVersion: config.istio.io/v1alpha2

kind: handler

metadata:

name: handler

namespace: istio-system

spec:

compiledAdapter: fluentd

params:

address: "fluentd-es.kube-system:24224"

---

# Rule to send logentry instances to the Fluentd handler

apiVersion: config.istio.io/v1alpha2

kind: rule

metadata:

name: newlogtofluentd

namespace: istio-system

spec:

match: "true" # match for all requests

actions:

- handler: handler

instances:

- newlog

---

Apply the below command

kubectl apply -f fluentd-istio.yaml

Filter on search with kubernetes.container.name is istio-proxy and we will see logs fron istio proxy.

## Some commands you may need

**Kubectl Commands**

|  |  |
| --- | --- |
| Command | Description |
| kubectl get ns | Get all namesapces |
| kubectl get svc -n istio-system | Get services under istio-system name space |
| kubectl get all -n istio-system | Get all under istio-system name space |
| kubectl delete ns istio-system | Delete namespace name istio-system |
| kubectl get all | Get everything in the kubernetes |
| kubectl delete --all pods | Delete all pods |
| kubectl delete --all pods –namespace = foo | Delete all pods under the namespace foo |
| kubectl delete --all deployments --namespace= foo | Delete all deployments under the namespace foo |
| kubectl delete --all namespaces | Delete all name spaces |
| kubectl delete --all svc | Delete all services |
| kubectl delete --all deployments | Delete all deployments |

**Docker Commands**

|  |  |
| --- | --- |
| Command | Description |
| docker rm -vf $(docker ps -aq) | To delete all containers including its volumes use |
| docker rmi -f $(docker images -aq) | To delete all the images |
| docker images | To check docker images |
| docker image build -t nexelus/location:1.0.1 | create a docker image name nexelus/location:1.0.1 |