**Microservices**

A diagram of a software project

Description automatically generated with medium confidence

**Common definitions:**

**YARP (Yet Another Reverse Proxy)**

is an open-source project by Microsoft designed to act as a high-performance **API Gateway** or **Reverse Proxy** in .NET applications. It's built on top of ASP.NET Core and allows for easy creation and customization of reverse proxies, including routing requests, load balancing, and more.

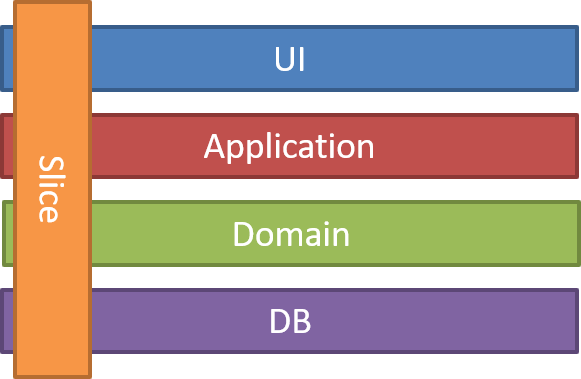
**Reverse proxy** is a type of server that sits between client requests and backend services. It acts as an intermediary, forwarding client requests to the appropriate backend server and returning the server's response to the client. This setup allows the reverse proxy to handle tasks like load balancing, security, and caching, enhancing the overall performance and security of your web applications or services.

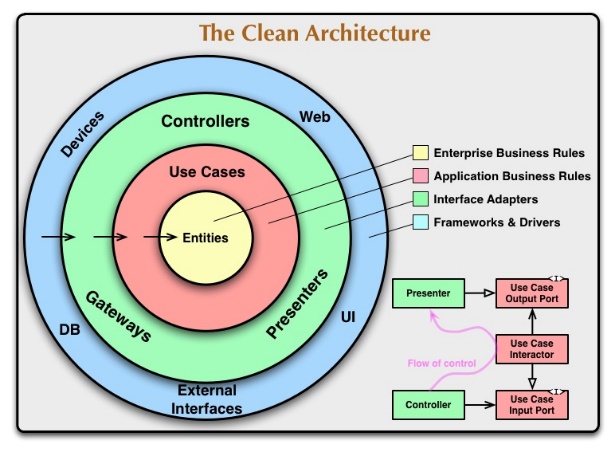
**Examples of Popular Reverse Proxies:**

* **NGINX**: Widely used reverse proxy server that provides load balancing, SSL termination, caching, and more.
* **HAProxy**: A high-performance reverse proxy known for its load balancing and reliability.
* **Apache HTTP Server**: Can be configured as a reverse proxy and used in various applications.
* **YARP (Yet Another Reverse Proxy)**: A modern reverse proxy built for .NET developers.

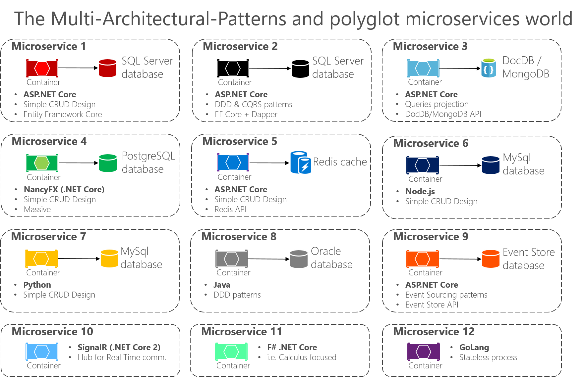
**Architectures**

**Layered Architecture (N-Tier Architecture)** The codebase is divided into horizontal layers based on technical concerns (e.g., UI, business logic, data access, etc.). These layers are organized so that each layer depends on the one below it, and functionality is distributed across multiple layers.

**Vertical Slice Architecture** is a design pattern in software development where the application is divided into distinct, independent "slices" that represent complete vertical segments of functionality, rather than being divided by technical layers (like UI, business logic, and data access layers). Each slice encapsulates all necessary components—UI, business logic, and data access—for a specific feature or use case, keeping them isolated from other slices.



**Clean Architecture** is a software design philosophy that emphasizes separation of concerns and independence of frameworks, tools, and external agencies. It aims to create a system where the business logic is decoupled from external concerns like UI, database, and frameworks, leading to more maintainable, flexible, and testable code.

**Multi-Architecture Polyglot Microservices** refers to a microservices architecture where different services (microservices) within the same system can use different programming languages, frameworks, or technologies. This approach allows for a diverse set of technologies to be utilized within a single application, enabling teams to choose the best tools for each specific task.

A diagram of a diagram

Description automatically generated

**Domain-Driven Design (DDD)** is a conceptual framework and set of principles for designing complex software systems by focusing on the core domain and its logic. it aims to bridge the gap between business needs and software design.

**Command Query Responsibility Segregation (CQRS)**

It is a pattern used in software architecture to separate the responsibilities of reading data from writing data, which can lead to more scalable and maintainable systems.

**gRPC** (gRPC Remote Procedure Calls)

is an open-source framework developed by Google that uses HTTP/2 for transport, Protocol Buffers (protobufs) for serialization, and supports multiple programming languages. It allows clients and servers to communicate with each other by calling methods as if they were local, but actually they are executed on a remote server.

**Comparison with REST:**

* **Protocol**: gRPC uses HTTP/2 while REST uses HTTP/1.1. HTTP/2’s features like multiplexing and header compression provide performance advantages for gRPC.
* **Serialization**: gRPC uses Protocol Buffers (binary) while REST often uses JSON (text-based). Protobufs are more compact and efficient compared to JSON.
* **Streaming**: gRPC has built-in support for streaming requests and responses, whereas REST typically uses a request-response model.
* **Performance**: gRPC is generally more efficient in terms of serialization and network usage, making it suitable for high-performance and low-latency applications.

**Use Cases:**

* **Microservices**: Ideal for communication between microservices in distributed systems due to its performance, support for streaming, and cross-language capabilities.
* **Real-time Communication**: Suitable for scenarios requiring real-time data streaming, such as live data feeds or chat applications.
* **High-Performance Systems**: Used in systems where performance and efficiency are critical, such as financial services or large-scale data processing.

A diagram of a service

Description automatically generated**What are Microservices?**

Microservices are small independent and loosely coupled services that can work together.

**Characteristics:**

1. Single responsibility
2. Independently Deployable
3. Decentralized Data Management (polyglot persistence)
4. Loose coupling
5. Lightweight communication
6. Scalability
7. Autonomous Development teams
8. Failure isolation
9. Technology agnostic
10. Continuous delivery & deployment
11. Event-Driven Architecture
12. Resilience & Fault Tolerance
13. Decentralized Governance
14. API Gateway
15. Service Discovery
16. Separation of Concerns

**Benefits:**

1. Faster Time to market
2. Scalability
3. Resiliency
4. Flexibility in technology
5. Improved maintainability

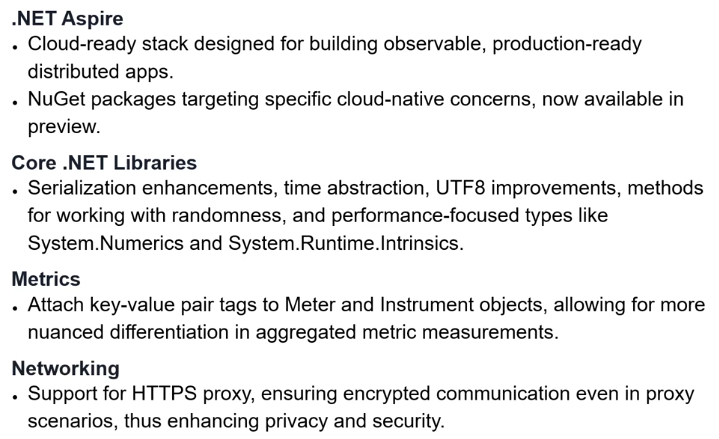
**Challenges:**

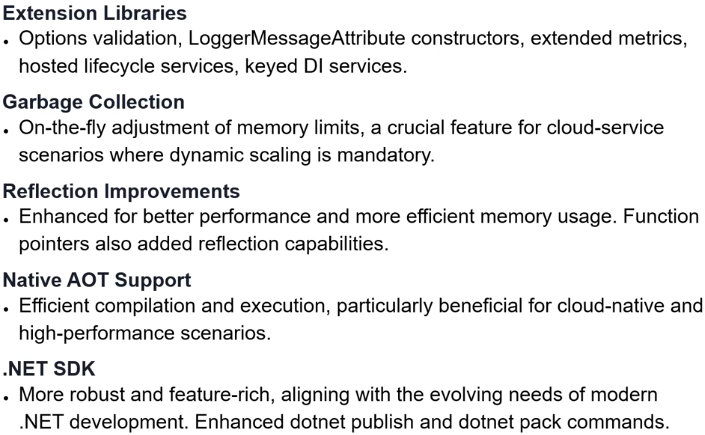
1. **Complexity**: Managing many services, especially with inter-service communication and data consistency, can be complex.
2. **Distributed Systems**: Microservices introduce distributed system challenges like latency, fault tolerance, and network issues.
3. **Data Consistency**: Ensuring consistency across distributed services and databases can be difficult.
4. **Operational Overhead**: Each service has its own lifecycle (building, deploying, monitoring, etc.), which can increase the operational load.

**Design patterns**

1. Database per Service
   * + Every service should have its own database. It can be polyglot persistence among microservices
     + The service’s database can’t be accessed directly by other microservices. Each service’s persistent data can only be accessed via Rest APIs.

**What is the new .Net 8?**





**What is new in C# 12:**

A black text on a white background

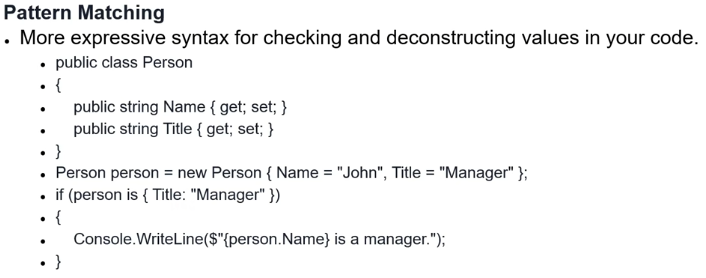
Description automatically generated

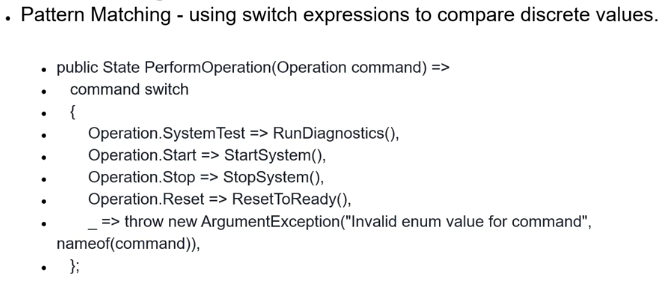
A screenshot of a computer program

Description automatically generated

A white background with black text

Description automatically generated





**Some Definitions:**

**Minimal APIs:** Refer to a streamlined approach for building simple, lightweight web APIs with minimal configuration and boilerplate code. Introduced in .NET 6, they are designed to make it easier and faster to create APIs with fewer complexities, while still leveraging the power of the full ASP.NET Core framework.

**What is Docker?**

* Docker is an open platform for developing, shipping, and running applications.
* Separate your applications from your infrastructure so you can deliver software quickly.
* Advantages of Docker's methodologies for shipping, testing, and deploying code quickly.
* Significantly reduce the delay between writing code and running it in production.
* Automating the deployment of applications as portable, self-sufficient containers that can run on the cloud or on-premises.
* Docker containers can run anywhere, in your local computer to the cloud.
* Docker image containers can run natively on Linux and Windows.
* Docker is defacto standard for containerization of microservices.
* Store images in registry, which is a library of images and is needed when deploying to production orchestrators
* **Orchestrating** whole **microservices** application with **Docker-Compose** for running multi-container Docker images. Single command, create and start all the services.

**Marten Library:**

**Marten** is a .NET library designed for working with PostgreSQL as a document store. It provides a simple and intuitive API for managing JSON documents, integrating with the .NET ecosystem seamlessly. Here are some of the features Marten offers:

1. **Document Storage**: Marten allows you to store and retrieve JSON documents directly in PostgreSQL. You can use C# objects, and Marten will handle the conversion to JSON automatically.
2. **Schema-less Design**: Similar to traditional document databases, Marten allows you to have a schema-less design, enabling you to store different structures of data in the same table.
3. **LINQ Support**: Marten supports LINQ queries, allowing developers to use familiar syntax to query their JSON documents.
4. **Multi-Tenancy**: Marten supports multi-tenant applications out of the box, allowing you to isolate data for different tenants while using the same database.
5. **Change Tracking and Event Sourcing**: Marten provides built-in support for change tracking and event sourcing patterns, making it easier to manage complex application states.

**MediatR Library:**

**MediatR** is a popular .NET library that facilitates the **Command-Query Responsibility Segregation (CQRS)** pattern by implementing the **Mediator design pattern**. It helps to decouple the sending and handling of requests, making it easier to organize application logic, especially in complex applications with multiple layers or microservices.

**Carter Library:**

**Carter** is a lightweight library for .NET that simplifies building HTTP APIs in .NET applications, particularly when used with ASP.NET Core. It provides a way to define **API endpoints in a modular, minimalistic, and organized way**, making it easier to keep your endpoint logic separate and maintainable.

**Mapster** **Library**:

**Mapster** is a high-performance, flexible object mapping library in .NET, primarily used to simplify the process of converting between different object types (such as DTOs, view models, and domain models). Mapster aims to be lightweight and fast, offering features for complex mappings without compromising on speed or maintainability.

**FluentValidation** **Library:**

**FluentValidation** is a popular .NET library for building strongly-typed, expressive validation rules for user input in a clean and readable way. It allows developers to create validation logic outside of the model classes, keeping code clean and following the single-responsibility principle. FluentValidation is often used in ASP.NET Core applications to validate request data in controllers or API endpoints.

**Building a Microservice:**

* When building a microservices application you should define each microservice **Port number** in the launchSettings.json the ASP.NET Core Ports are listed as http / https   
  A close-up of a computer

  Description automatically generated