One Day
Rethink application

Pattern e Servizi per Azure ma senza dimenticare l'on premise

Marco Parenzan

un ringraziamento a

Who am I

MARCO PARENZAN

beanTech
IT moves your business

MVP Microsoft® Most Valuable Professional

innOva community

WPC

marco_parenzan

marcoparenzan

marcoparenzan

marcoparenzan

Pattern e Servizi per Azure
ma senza dimenticare l'on premise

- Considerazioni fatte a seguto dello sviluppo di una applicazione che doveva girare in Azure e on premise minimizzando lo sforzo
- Considerazioni sulla percezione (e sponsorizzazione) di Azure grazie all'on premise
- Considerazioni sull'on premise in un era cloud

La speranza dell'utility computing

Nel 2010...

- Nell'era del web, i provider
- Un nuovo hosting...
- Macchine virtuali e servizi cloud
- Windows AzureUtility Computing

...e un sacco di entusiasmo!

Alcune verità su Azure (e sul cloud in generale) per il 2024

- Piace
- Interessa
- I client lo chiedono (?)

...ma...

Ma…(il cloud…)

- Nel breve…
- Nella non evoluzione….
- Nella non crescita…
- Nella non competizione…

…(il cloud) costa!

Il costo…

- …non è solo monetario…
- …è anche di operations (per dire che quando fai un lift&shift non c'è solo il risparmio del ferro, ma anche il costo di non avere totalmente il controllo)
- …e i limiti introdotti senza adeguamento (una VM è 99.9% disponibile…Azure la può far ripartire – quasi – quando vuole) ne abbassano il valore

Le aziende (clienti e fornitori) non hanno imparato…

- …che Il cloud è una enorme piattaforma di operations…
- …ad avere veramente bisogno della scalabilità e dell'elasticità
- …a capire bene la differenza tra CAPEX e OPEX
- …non hanno imparato a calcolare i costi nascosti
- Il vero motivo per cui il cloud viene creato

Un business in continua evoluzione…
…perchè è il mercato che lo vuole…
…almeno…fuori dall'italia è così…

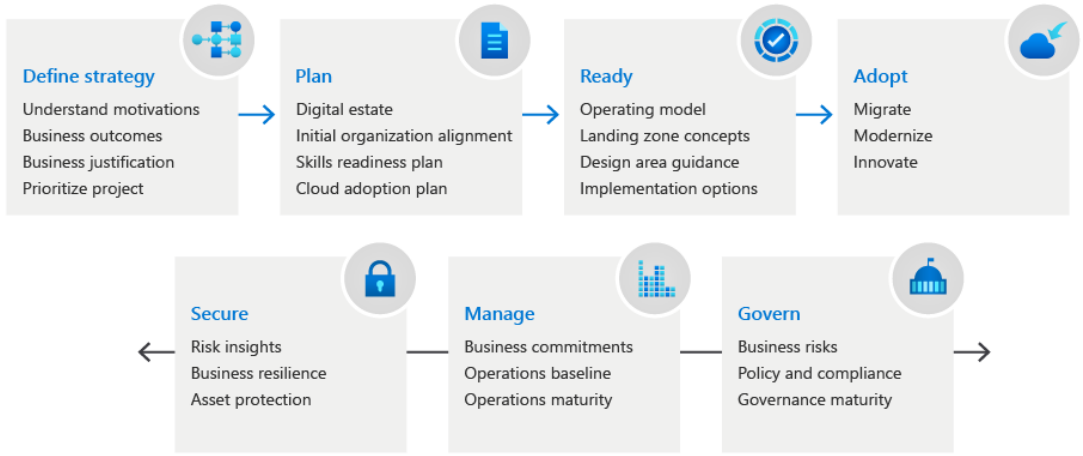
Cosa significa “aver imparato”?

- Il cloud non serve per fare ottimizzazione dei costi (o almeno io non l’ho ancora visto…e comunque direi che certamente non è quello che succede in Italia)
- Il cloud serve a supportare la creazione di nuovo business…
- …continuamente…
- …e continuamente fare App Innovation…
- …e refactoring…
- ..perchè dal cloud abbiamo imparato tanto!

Pensare alle applicazioni nell’era cloud

Cos’è il Cloud Adoption Framework

- Cloud Adoption Framework (CAF) è una guida completa che copre l'intero percorso verso il cloud, dalla strategia alla governance. Fornisce procedure consigliate, strumenti e linee guida per ogni fase del ciclo di vita dell'adozione del cloud: definire la strategia, pianificare, preparare, adottare, governare e gestire. Il CAF ti aiuta ad allineare i tuoi obiettivi aziendali con la tua visione cloud, valutare la tua preparazione, preparare il tuo ambiente, implementare le tue soluzioni e ottimizzare le tue operazioni.



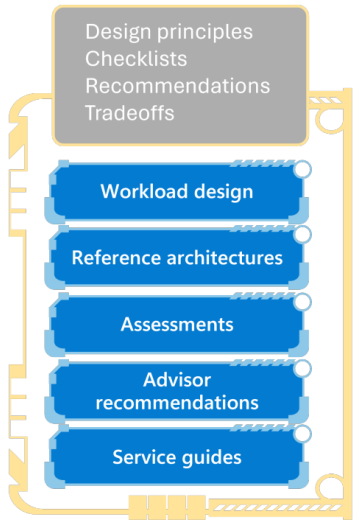
https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/

Perchè vorremmo andare in cloud?

Critical business events	Migration	Innovation
Datacenter exit	Cost savings	Preparation for new technical capabilities
Merger, acquisition, or divestiture	Reduction in vendor or technical complexity	Building new technical capabilities
Reduction in capital expenses	Optimization of internal operations	Scaling to meet market demands
End of support for mission-critical technologies	Increase in business agility	Scaling to meet geographic demands
Response to regulatory compliance changes	Preparation for new technical capabilities	Improved customer experiences and engagements
New data sovereignty requirements	Scaling to meet market demands	Transformation of products or services
Reduction of disruptions and improvement of IT stability	Scaling to meet geographic demands	Market disruption with new products or services
Report and manage the environmental impact of your business	Integration of a complex IT portfolio	Democratization and/or self-service environments

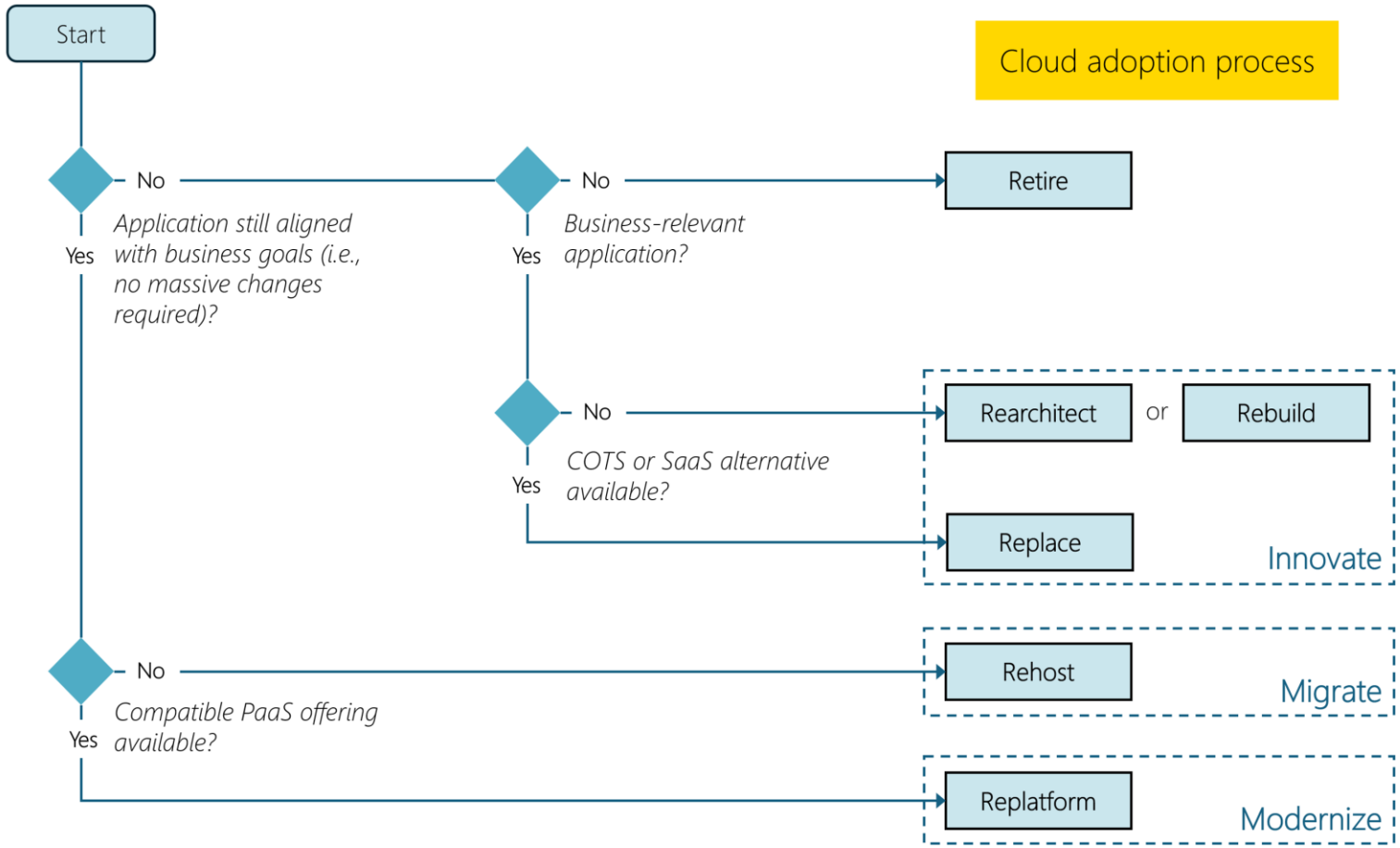
Cos’è il Well-Architected Framework

- Il Well-Architected Framework (WAF) è un insieme di principi e linee guida che consentono di progettare e gestire sistemi affidabili, sicuri, efficienti e convenienti nel cloud. Si compone di cinque pilastri: eccellenza operativa, sicurezza, affidabilità, efficienza delle prestazioni e ottimizzazione dei costi. Il WAF consente di valutare l'architettura in base a questi pilastri e di identificare le aree di miglioramento. Fornisce inoltre le migliori pratiche e raccomandazioni per ciascun pilastro.

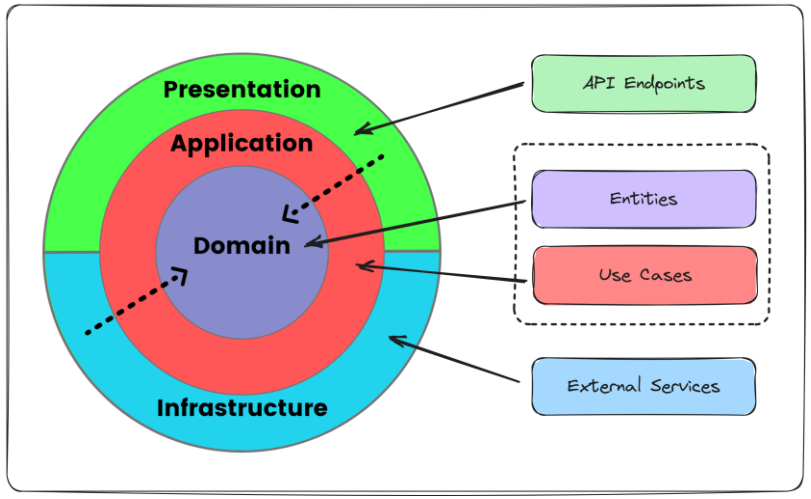


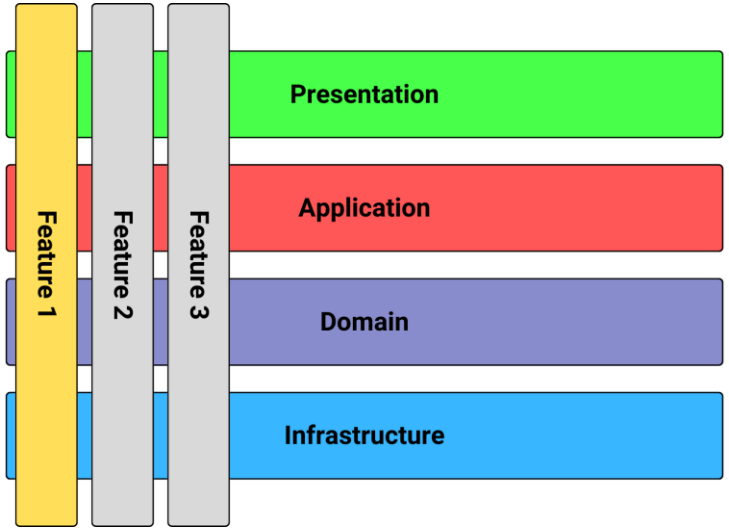
https://learn.microsoft.com/en-us/azure/well-architected/

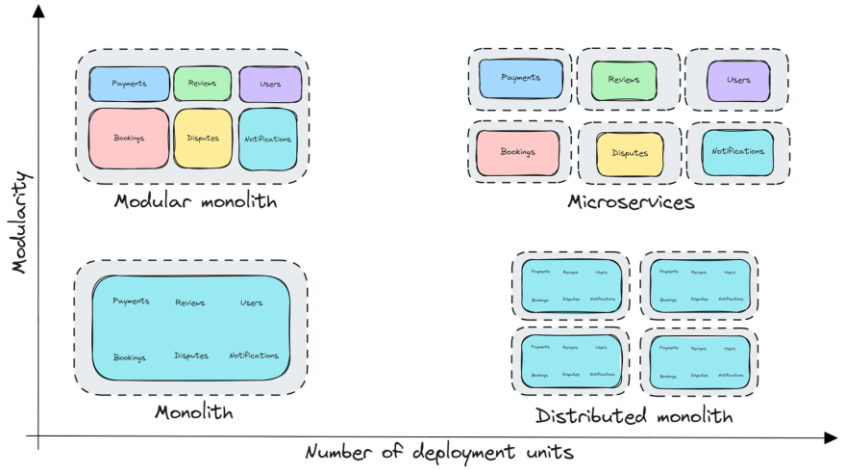
Prendiamo il Cloud Adoption Process dal CAF



Non parliamo di scrittura del codice di per sé







A Journey from monolith to microservices
Simone Recupero
Christian De Simone

…ma mi sento di affermare che…

- …le considerazioni che possiamo fare per qualunque di queste architetture vanno bene ovunque le deployamo…
- …in cloud…
- …on premise…
- …qui ci mettiamo “il nostro codice”…cioè quello che scriviamo…
- …il nostro compute!

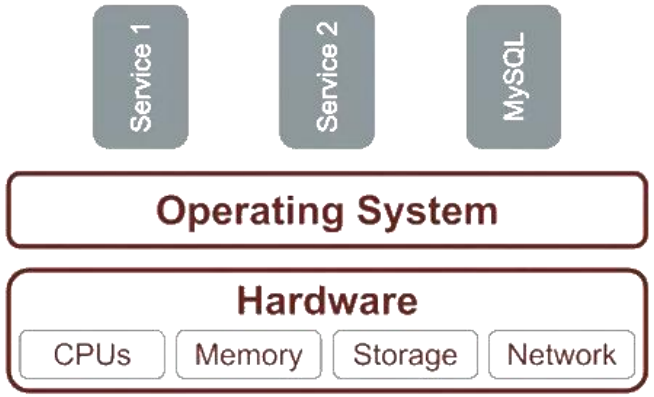
Deviazione…torneremo in questo punto fra poco

Il compute nell'era della containerizzazione

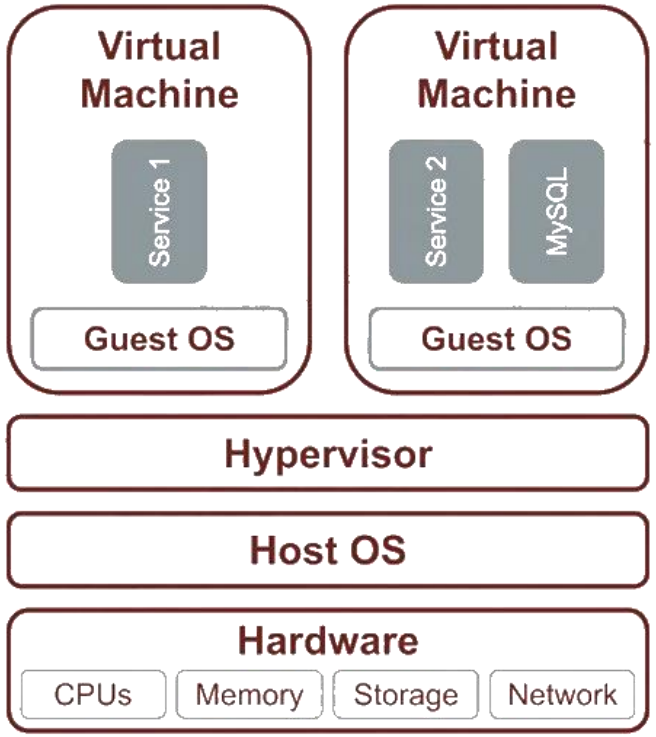
Virtual Machines vs. Containers

Virtual Machine
A virtual computer running in a computer using the hardware resources isolated and partitioned from the host.

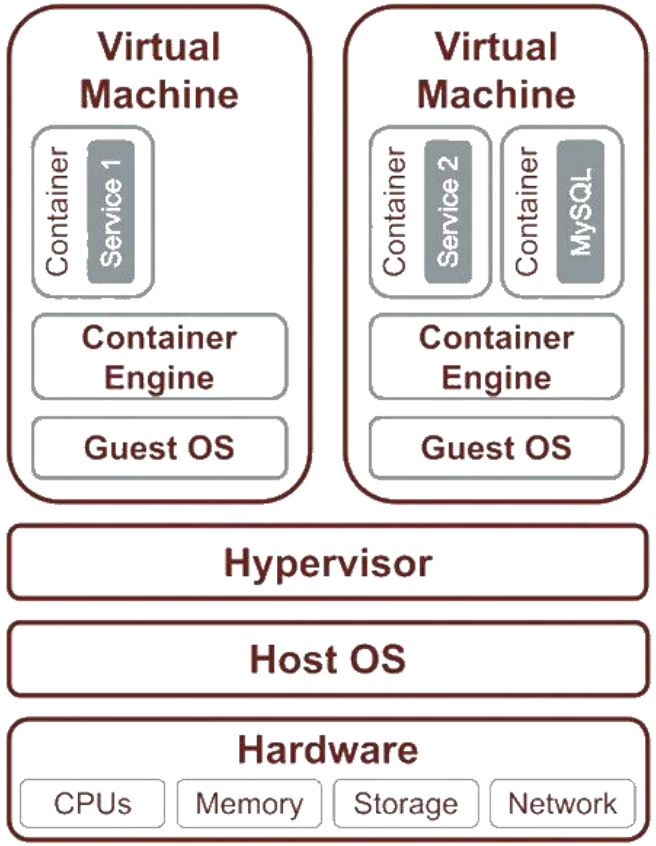
Container
Lightweight packages of the application code along with all the dependencies required to run the services.



Bare Metals
Services run directly on the physical machine

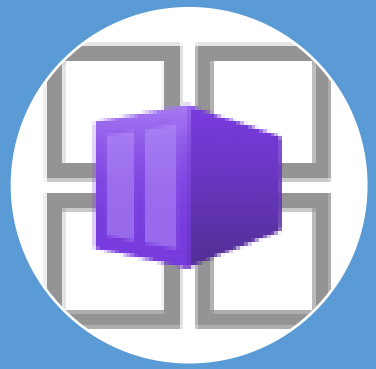


Virtual Machines
One or more services run in each virtual machine

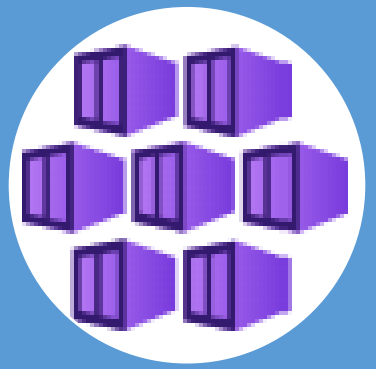


Containers in VMs
Each service runs in a separate container, which is running in a VM

Tipi di servizi container



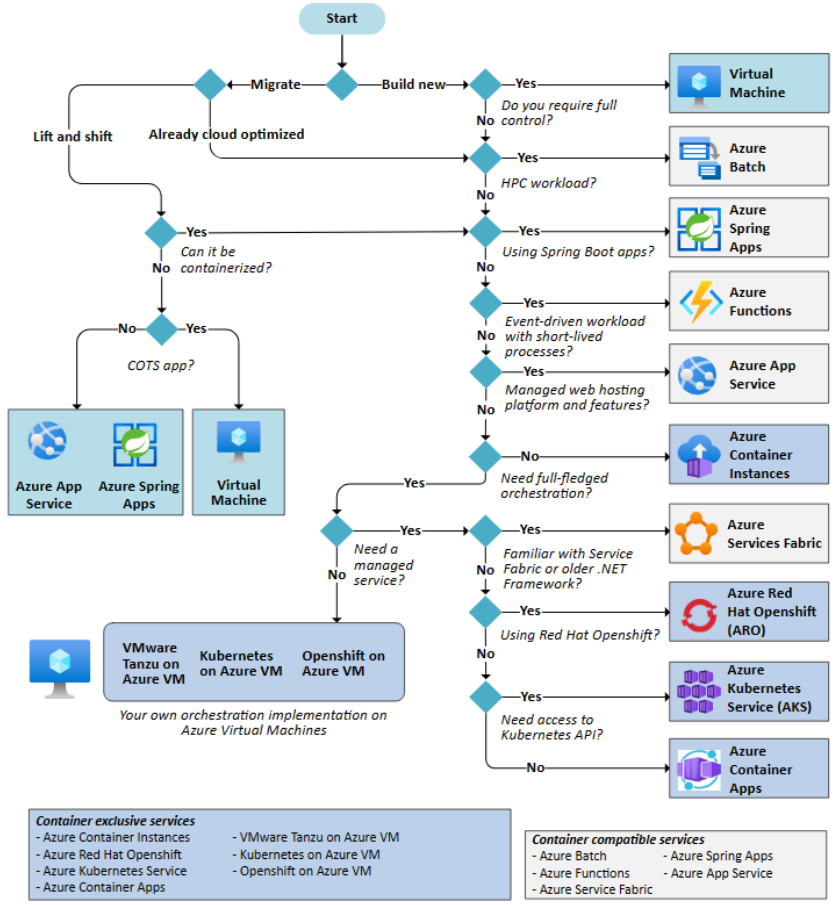
Azure Container Apps is a fully managed Kubernetes-based application platform that helps you deploy HTTP and non-HTTP apps from code or containers without orchestrating infrastructure. For more information, see Azure Container Apps documentation.



Azure Kubernetes Service (AKS) is a managed Kubernetes service for running containerized applications. With AKS, you can take advantage of managed add-ons and extensions for additional capabilities while preserving the broadest level of configurability. For more information, see AKS documentation.

Web App for Containers is a feature of Azure App Service, a fully managed service for hosting HTTP-based web apps with built-in infrastructure maintenance, security patching, scaling, and diagnostic tooling. For more information, see App Service documentation.

Scegliamo un compute service in Azure



Microsoft Azure

Considerazione sui container

- Riconosco i valori dei container, ma…
- …non li ritengo vincolanti
  - W Docker Desktop a pagamento!
- Il mondo industriale ad esempio è un mondo Windows…
  - …fermo restando che oramai amo Windows, ma lo ritengo limitante
- …ma comunque tante aziende parlano di Windows e non hanno fatto un passaggio a container
- …fermo restando che il miglior open source è su Linux e il container è la morte sua…
- Senza parlare di Linux (anche se mi pare che parlare di Linux senza container non sia il target…)
- Fermo restando che se si può…meglio passare a container!

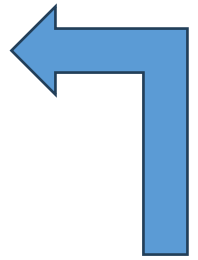
I dubbi sulle Azure Function

- Le functions sono un servizio fantastico…
- …sicuri?
- La mia affermazione: le functions sono “webhook”…
- …cioè non mi piacciono i bindings….
  - (meglio Event Grid)
- Non è che piacciono perchè sono serverless?
  - E adesso ci sono le container apps (che obbligano ai container…è vero…)
- A me è piaciuto molto il passaggio con .NET 8 ad un full isolated mode. Perchè?
- Perchè è bene che siano un processo standard…
- …ed è anche Ottimo che le functions sostanzialmente non siano più diverse da altri processi ASP.NET core-like
- Le function possono essere comunque portabili on premise…
- …ma onestamente on premise preferisco portare un processo ASP.NET

Comunque spostate la logica in una library

- Spostare la logica per quanto possibile fuori dal progetto Host
- Incapsulare in maniera corretta le dipendenze➔minimizzare
- Attenzione a librerie «tecnologiche» e librerie di «modello»

Ciò che non sta nel “compute”



Torniamo dalla deviazione

Stavamo dicendo "...ma mi sento di affermare che..."

- ...le considerazioni che possiamo fare per qualunque di queste architetture vanno bene ovunque le deployamo...
- ...in cloud...
- ...on premise...
- ...qui ci mettiamo "il nostro codice"...cioè quello che scriviamo...

Ma come gestiamo il codice non nostro?

- Abbiamo due livelli
- Librerie…
- …servizi terzi…
- Anche qui una semplificazione con una affermazione…
- …”le librerie vanno bene comunque, on premise e in cloud”
- Il ragionamento che faccio dopo si può applicare anche alle librerie, ma è…meno strategico…comunque tenetene conto alla fine…

Quindi cosa manca?

Mancano i servizi…

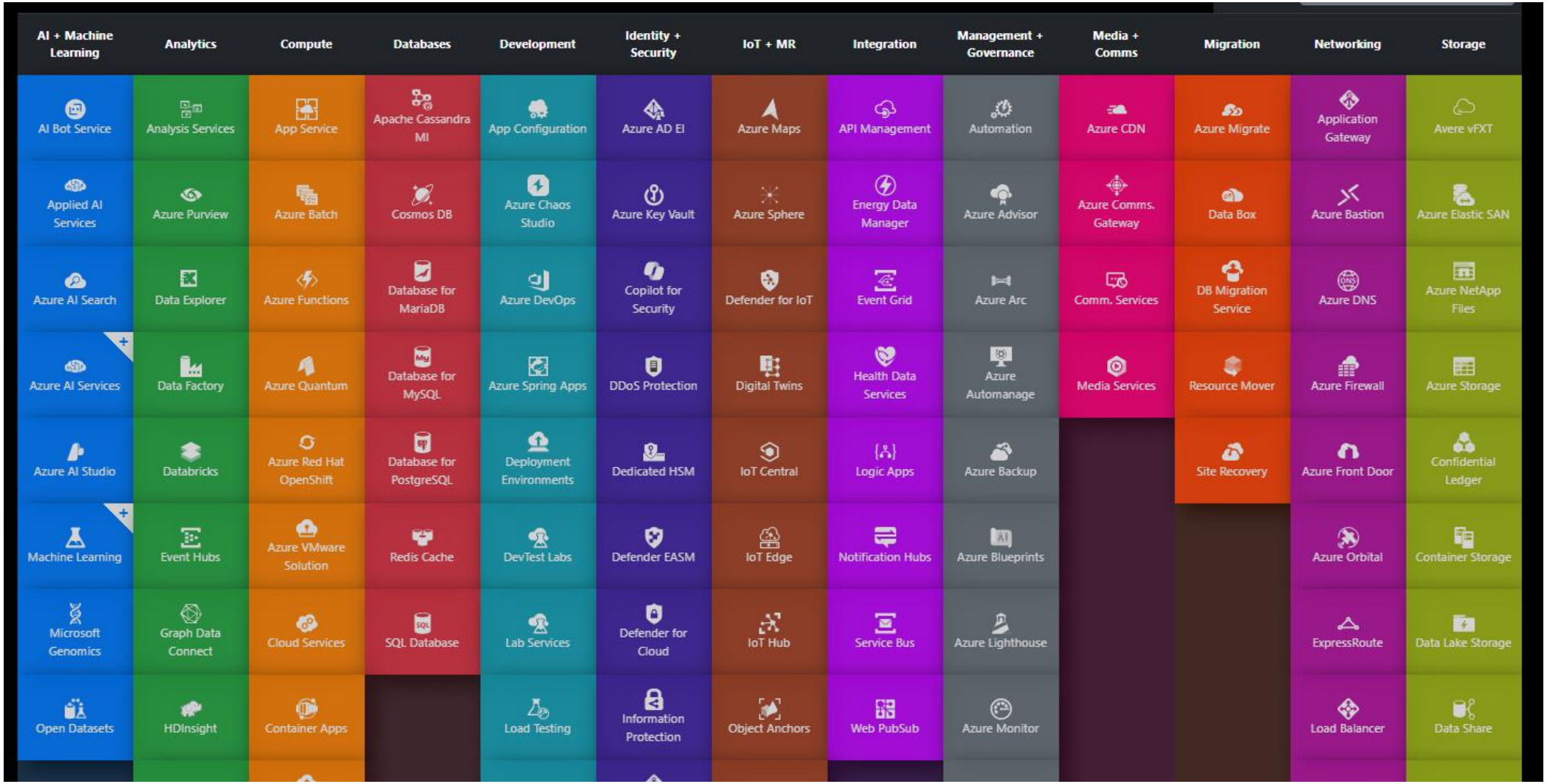
- … che usiamo “out of the box”..
- …insomma…dal db sql in poi…
- …chiaro no?

Affermazione

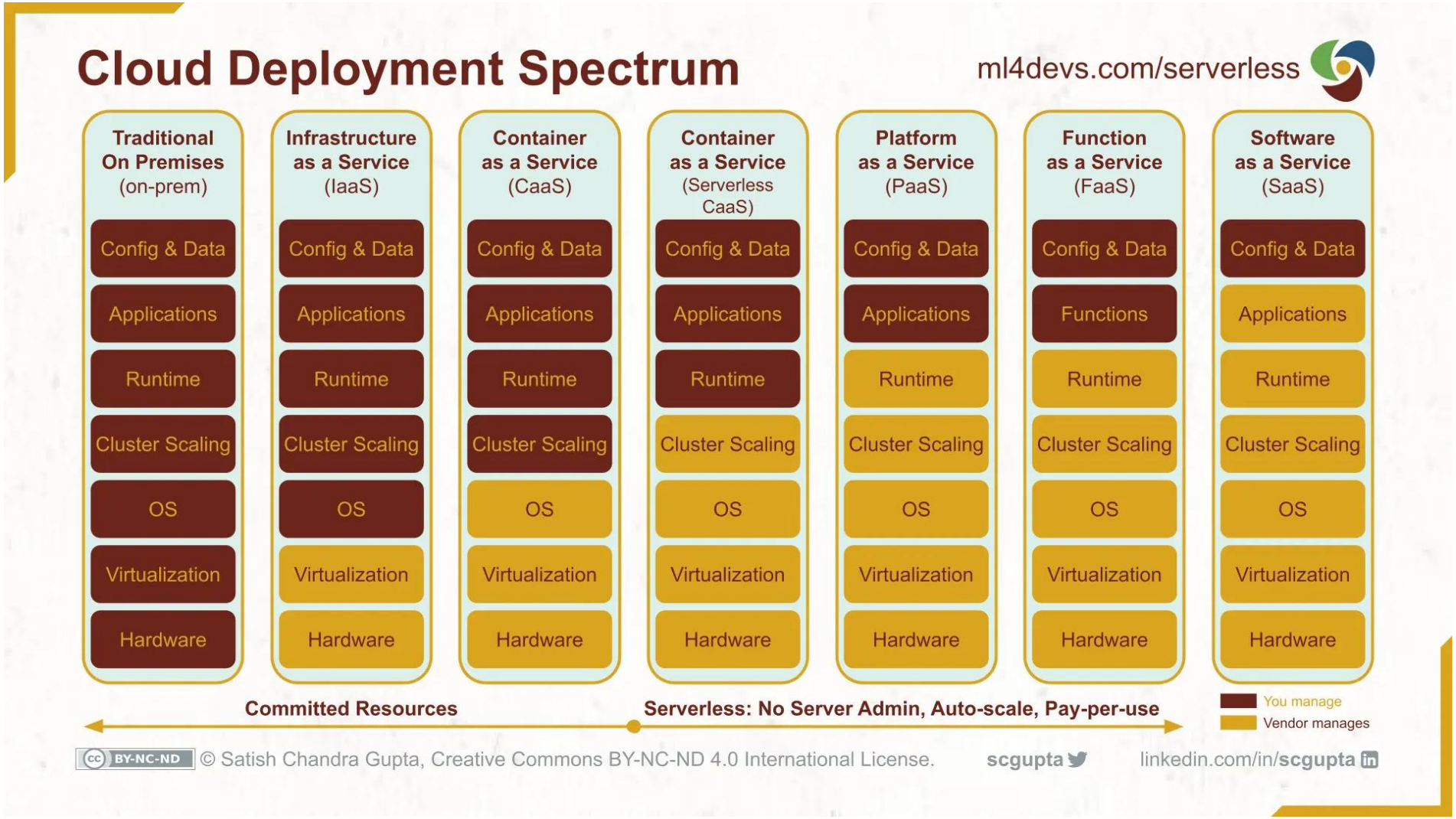
- È impossibile sviluppare per il cloud senza tenere conto dell'on premise
- Parti di codice e/o servizi si vogliono riusare
- È invece meno probabile il multicloud
  - Sebbene può essere comunque una opportunità

Servizi cloud
e la necessità dell’on-premise

https://azurecharts.com/overview



Shared responsibility model



Considerazioni sui servizi Azure (e in generale cloud)

- Servizi solo PaaS/SaaS e funzionalmente distanti…
- Servizi solo PaaS/SaaS ma funzionalmente simili
- Servizi esplicitamente equivalenti

Servizi solo PaaS/SaaS e funzionalmente distanti…

- Alcuni esempi
  - IoT Hub
  - Cosmos Db
  - Fabric/Power BI
  - Adx
- Criticità
  - Funzionalità molto caratteristiche
  - Molto “protocol oriented”
  - Similitudini rischiose (più da company o da integrator che da ISV)

Servizi solo PaaS/SaaS ma funzionalmente simili

- Alcuni esempi
  - Service Bus
  - Storage Account/Blob
  - File System
- Equivalenti
  - RabbitMQ
  - Min.io
  - Windows File System (per usare l'esempio SMB)
- Criticità
  - Alcune funzionalità sono molto allettanti ma non portabili

Servizi esplicitamente equivalenti

- Alcuni esempi
  - Redis
  - Postgres
- Criticità
  - Beh…a questo livello…nessuna…

I nostri amati RDBMS

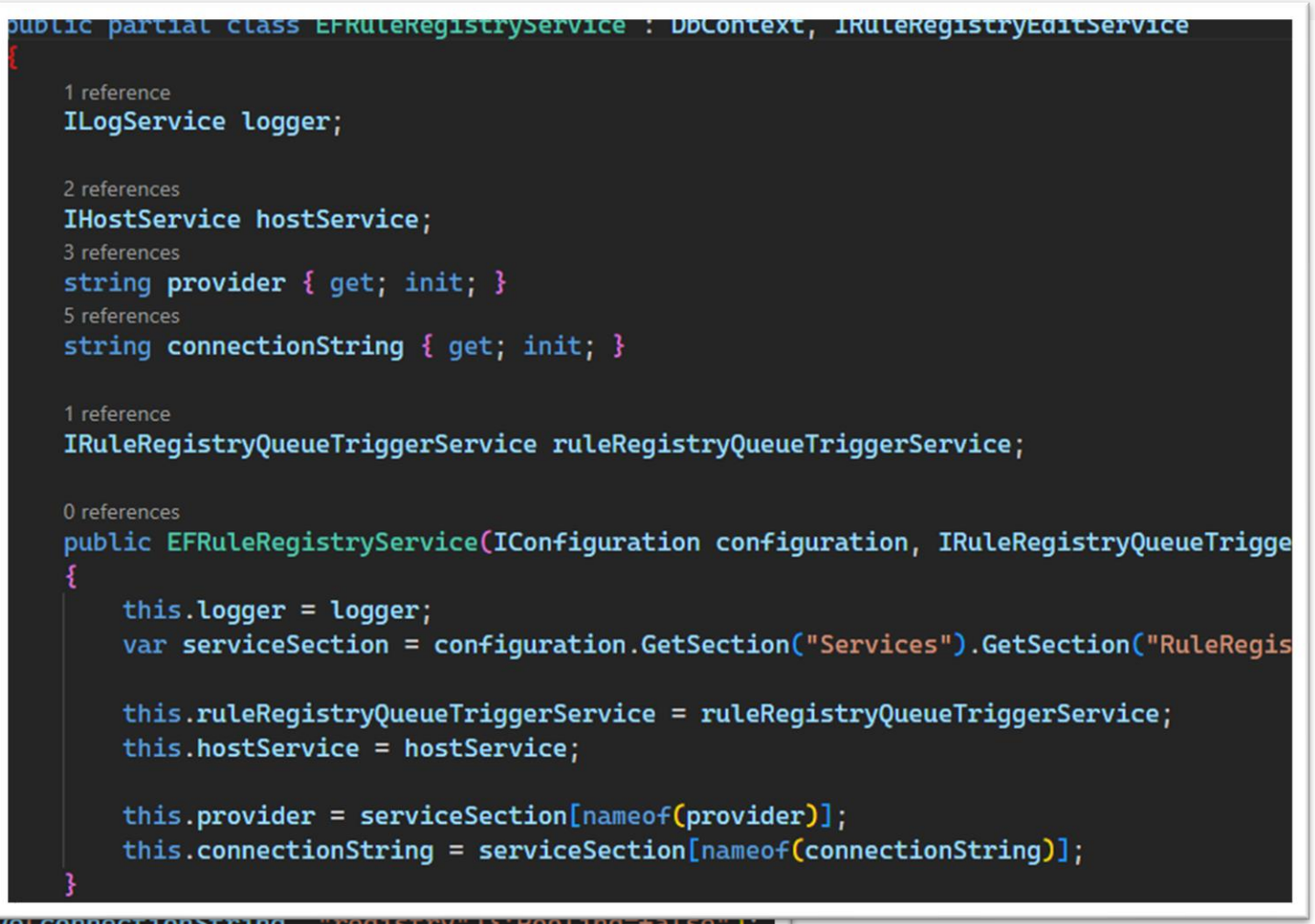
- SQL Server e Postgres sono I nostri dbms relazionali preferiti
- Azure SQL Server e Azure SQL for Postgres sono I servizi in cloud
- Postgres in Container (anche SQL ma a me non è che mi esalti eh?)
- Funzionali ed equivalenti anche in PaaS (senza entrare nel merito di SQL Managed Instances)
- Se sei un ISV su base .NET, Entity Framework è il tuo strumento di riferimento.
  - Dapper onestamente no, nonostante le performance (indiscutibili) ma bisogna favorire la portabilità
  - Sempre che non si faccia onestamente uso di stored procedures
  - Ma vedo tanti che non lo fanno e/o non lo faranno mai
- Dapper è più uno strumento da integratore per il caso specific (ma non voglio fare polemica eh?)
  - EF più utile come prodotto.

EF, DbContext e DDD

- il singolo DbContext che è “immagine” di tutto il db…ma anche no
  - Pesante da (s)caricare in un contest condiviso in cui è necessario (s)caricare oggetti velocemente
- Se prendiamo spunto dal DDD, dal bounded context, dai vertical slices e/o modular monoliths…
  - Tanti DbContext possono diventare degli Aggregate Root, con le entità private e metodi pubblici che leggono/scrivono DTO
- Ovvero…portiamo il nostro Db verso degli oggetti veri e propri…un po’ più distanti rispetto a un db

Some sample code...

```
protected override void OnConfiguring(DbContextOptionsBuild
{
    switch (provider)
    {
        case "mssql":
            optionsBuilder
                .UseSqlServer(connectionString);
            break;
        case "npgsql":
            optionsBuilder
                .UseNpgsql(connectionString);
            break;
        case "sqlite":
            optionsBuilder
                .UseSqlite($"Data Source={hostService.Resolve(connectionString, "registry")};Pooling=false");
            break;
    }
}
```



```
public partial class EFRuleRegistryService : DbContext, IRuleRegistryEditService
{
    1 reference
    ILogService logger;

    2 references
    IHostService hostService;
    3 references
    string provider { get; init; }
    5 references
    string connectionString { get; init; }

    1 reference
    IRuleRegistryQueueTriggerService ruleRegistryQueueTriggerService;

    0 references
    public EFRuleRegistryService(IConfiguration configuration, IRuleRegistryQueueTrigge
    {
        this.logger = logger;
        var serviceSection = configuration.GetSection("Services").GetSection("RuleRegis

        this.ruleRegistryQueueTriggerService = ruleRegistryQueueTriggerService;
        this.hostService = hostService;

        this.provider = serviceSection[nameof(provider)];
        this.connectionString = serviceSection[nameof(connectionString)];
    }
```

Come usare servizi cloud e on premise in maniera trasparente

Rispolveriamo (?) vecchie conoscenze...

- Patterns: tre generazioni (con overlap ☺)
  - Design Patterns
  - Enterprise Design Patterns (https://martinfowler.com/eaaCatalog/)
  - Cloud Design Patters
- Pattern: software/services implementano patterns
  - Servizi equivalenti cloud/on premises
  - https://learn.microsoft.com/en-us/azure/architecture/
  - https://learn.microsoft.com/en-us/azure/architecture/patterns/

I pattern e i servizi cloud

- I servizi Azure sono implementazioni di pattern
- I software sono implementazioni di pattern
- Preferisco i servizi Azure non per TUTTI gli aspetti funzionali
  - Prendo solo gli aspetti che intersecano con i servizi on premise
- Preferisco i servizi Azure per gli aspetti non funzionali
  - Alta disponibilità
  - Resilienza
  - Continuità
  - Backup
  - Operations

GoF Design Patterns

Creational:	Structural:	Behavioral:
• Factory Method • Abstract Factory • Prototype	• Façade • Adapter • Composite	• Observer • Strategy • State

Pattern Creational

- I pattern creazionali sono una categoria di design patterns che si occupano della creazione di oggetti, astraendo il processo di istanziazione in modo che il sistema sia indipendente da come gli oggetti vengono creati, composti e rappresentati.
- Questi pattern sono utili per gestire la creazione di oggetti in modo flessibile e riutilizzabile, riducendo le dipendenze e facilitando la manutenzione del codice.

- Abstract Factory: Fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete.
- Separa la costruzione di un oggetto dalla sua rappresentazione, in modo che lo stesso processo di costruzione possa creare diverse rappresentazioni.
- Factory Method: Definisce un'interfaccia per creare un oggetto, ma lascia che le sottoclassi decidano quale classe istanziare. Il Factory Method consente a una classe di delegare l'istanziazione alle sottoclassi.
- Crea nuovi oggetti clonandone uno esistente, consentendo di aggiungere o rimuovere proprietà all'oggetto clonato senza modificare l'originale.
- Singleton: Assicura che una classe abbia solo una istanza e fornisce un punto di accesso globale a quell'istanza.

Pattern Strutturali

- I pattern strutturali sono un gruppo di design patterns che si occupano di come le classi e gli oggetti vengono composti per formare strutture più grandi. I pattern strutturali semplificano la struttura fornendo modi per identificare relazioni semplici tra entità, in modo che il sistema sia più facile da capire e da lavorare.

- **Adapter**: Consente a interfacce incompatibili di lavorare insieme. Agisce come un ponte tra due interfacce incompatibili.
- **Bridge**: Separa l'astrazione dall'implementazione in modo che le due possano variare indipendentemente.
- **Composite**: Consente di trattare singoli oggetti e composizioni di oggetti in modo uniforme.
- **Decorator**: Aggiunge responsabilità aggiuntive a un oggetto in modo dinamico.
- **Façade**: Fornisce un'interfaccia unificata a un insieme di interfacce in un sottosistema, rendendo il sottosistema più facile da utilizzare.
- **Flyweight**: Riduce il costo della creazione e manipolazione di un gran numero di oggetti simili.
- **Proxy**: Fornisce un surrogato o segnaposto per un altro oggetto per controllare l'accesso ad esso.

Pattern Strutturali

Pattern Façade

- Il pattern Façade è un design pattern strutturale che fornisce un'interfaccia semplificata a un insieme di interfacce in un sottosistema, rendendo il sottosistema più facile da utilizzare.
- Il pattern Façade può essere implementato creando una classe 'Facade' che contiene un insieme di membri che delegano le chiamate ai componenti del sottosistema appropriati.
- Questo permette di isolare la complessità del sottosistema dai clienti e di promuovere un accoppiamento debole tra il sottosistema e i suoi clienti.

Pattern Proxy

- Il pattern Proxy in C# è un design pattern strutturale che fornisce un oggetto surrogato o segnaposto per un altro oggetto per controllare l'accesso a quest'ultimo.
- Questo è utile per gestire operazioni costose o complesse quando si accede a un oggetto, per fornire una livello aggiuntivo di sicurezza, o per aggiungere funzionalità aggiuntive durante l'accesso all'oggetto.

Differenza tra Façade e Proxy

- Il pattern **Façade** e il pattern **Proxy** sono entrambi pattern strutturali, ma hanno scopi e implementazioni differenti:
  - **Facade**: Questo pattern è utilizzato per creare un'interfaccia semplice per un sottosistema complesso. Il suo obiettivo principale è semplificare l'interazione tra il client e il sottosistema. Il pattern Facade non nasconde le classi del sottosistema; piuttosto, fornisce un'interfaccia unica che ridirige le chiamate ai metodi appropriati all'interno del sottosistema. È come una reception che aiuta a indirizzare le tue richieste al dipartimento giusto all'interno di un'organizzazione.
  - **Proxy**: Questo pattern fornisce un sostituto per un altro oggetto per controllare l'accesso a esso. Il Proxy può aggiungere funzionalità come il controllo dell'accesso, il caricamento pigro, o il logging, senza modificare il codice dell'oggetto originale. È come un intermediario che controlla l'accesso a una risorsa, potenzialmente eseguendo azioni aggiuntive quando quella risorsa viene richiesta.
- In sintesi, il **Façade** semplifica l'interfaccia di un sottosistema complesso per i clienti, mentre il **Proxy** controlla e gestisce l'accesso a un oggetto, offrendo la possibilità di interporre azioni prima o dopo l'accesso all'oggetto target. Entrambi i pattern contribuiscono a ridurre la complessità e a migliorare la struttura del codice, ma lo fanno in modi diversi e per scopi diversi.

Pattern Behavioural

- I pattern comportamentali (Behavioral Patterns) in C# sono una categoria di design patterns che si occupano di come gli oggetti interagiscono e si distribuiscono le responsabilità. Questi pattern sono particolarmente utili per gestire algoritmi, relazioni tra oggetti e le responsabilità tra oggetti comunicanti.
- Questi pattern aiutano a gestire algoritmi, relazioni e responsabilità tra oggetti, rendendo il codice più flessibile e riutilizzabile.

Pattern Behavioural

- **Chain of Responsibility**: Passa una richiesta lungo una catena di handler. Quando un oggetto riceve una richiesta, può sia gestirla sia passarla al prossimo handler nella catena.
- **Command**: Incapsula una richiesta come un oggetto, consentendo di parametrizzare i client con code, richieste e operazioni.
- **Interpreter**: Definisce una rappresentazione grammaticale per una lingua e un interprete che usa la rappresentazione per interpretare le espressioni nella lingua.
- **Iterator**: Fornisce un modo per accedere agli elementi di un oggetto aggregato sequenzialmente senza esporre la sua rappresentazione sottostante.
- **Mediator**: Definisce un oggetto che incapsula come un insieme di oggetti interagisce. Il pattern Mediator promuove il disaccoppiamento riducendo le comunicazioni dirette tra oggetti.
- **Memento**: Senza violare l'incapsulamento, cattura e esternalizza lo stato interno di un oggetto in modo che l'oggetto possa essere ripristinato in questo stato in seguito.
- **Observer**: Definisce una dipendenza uno-a-molti tra oggetti in modo che quando un oggetto cambia stato, tutti i suoi dipendenti vengono notificati e aggiornati automaticamente.
- **State**: Consente a un oggetto di alterare il suo comportamento quando il suo stato interno cambia. L'oggetto sembrerà cambiare la sua classe.
- **Strategy**: Definisce una famiglia di algoritmi, incapsula ciascuno e li rende intercambiabili. La strategia consente all'algoritmo di variare indipendentemente dai client che lo utilizzano.
- **Template Method**: Definisce lo scheletro di un algoritmo in un metodo, posticipando alcuni passaggi alle sottoclassi. Il Template Method consente alle sottoclassi di ridefinire alcuni passaggi di un algoritmo senza cambiare la struttura dell'algoritmo.
- **Visitor**: Rappresenta un'operazione da eseguire sugli elementi di una struttura oggetto. Il Visitor consente di definire una nuova operazione senza cambiare le classi degli elementi su cui opera.

OOP to the resque

- Il polimorfismo con le interfacce permette di definire un contratto che può essere implementato da diverse classi in modi diversi.
  - Ogni classe che implementa l'interfaccia deve fornire un'implementazione concreta per tutti i metodi definiti nell'interfaccia.
  - Questo consente agli oggetti di essere trattati come istanze dell'interfaccia piuttosto che come istanze di una particolare classe, facilitando così la scrittura di codice più flessibile e riutilizzabile.
- In realtà questo è un approccio «agile», da user story
  - L'interfaccia pubblica deve essere Iqueryable?
  - Oppure è corretto definire tutte e sole le operazioni possibili, senza «generalismi strani», implementati da metodi chiari e precisi?
- Questo è plausibile ovviamente in termini di OOP appunto con Interfacce e Polimorfismo (e una implementazione, nel caso del DB, con una classe base "tecnologica")

(ASP.NET) Dependency Injection

- La Dependency Injection (DI) in .NET è un pattern di design che consente di ridurre l'accoppiamento tra le classi e aumentare la modularità del codice. DI permette di iniettare le dipendenze in una classe, piuttosto che farle creare direttamente dalla classe stessa. Questo facilita la gestione delle dipendenze, specialmente in applicazioni di grandi dimensioni, e rende il codice più testabile e manutenibile.
- In .NET, DI è spesso implementata utilizzando un container DI, che è responsabile della creazione degli oggetti e della gestione delle loro dipendenze. Quando una classe richiede un oggetto, il container DI fornisce una istanza di quell'oggetto, insieme a tutte le dipendenze necessarie.

DEMO

Conclusioni

Conclusioni

- Quanto costa supportare contemporaneamente Azure e On Premise?
  - Dipende
  - Usare feature specifiche (ma anche qui dipende)
- Quanto si guadagna a supportare contemporaneamente Azure e On Premise?
  - Codice più pulito
  - Codice molto più strutturato
  - Costo iniziale più elevato, ma si guadagna dopo!
- Cloud e On premise si aiutano a vicenda!

Grazie

Domande?

marco_parenzan

marcoparenzan

marcoparenzan

marcoparenzan