Final Report

Andrea Rettaroli
andrea.rettaroli@studio.unibo.it

Luglio 2021

Nel 2016 il mercato delle architetture Serverless aveva un valore di 1,9 miliardi di dollari, nel 2020 il valore è arrivato a 7,6 milardi e si prevede che nel 2021 arriverà a 21,1 miliardi. Ad oggi le più grandi aziende di tecnologie: Google, Amazon, Microsoft, all'interno delle rispettive piattaforme cloud: Google Cloud, AWS, Azure; hanno messo a disposizione servizi che permettono lo sviluppo secondo architetture "Backend as a Service" (BaaS) e "Functions as a Service" (FaaS) e hanno visto crescere notevolmente il loro fatturato grazie ad esse. Con l'avvento del cloud questa nuova forma di sviluppo e architetture ha preso fortemente piede perchè permette di scorporare le attività di routine per il provisioning, la manutenzione e la scalabilità dell'infrastruttura server che vengono gestite da un provider di servizi cloud. Gli sviluppatori devono semplicemente preoccuparsi di creare pacchetti di codice da eseguire all'interno di container remoti o di progettare le loro soluzioni icludendo e sfruttando i servizi messi a disposizione dai cloud provider. Vedremo, grazie a degli esempi pratici, come questo nuovo paradigma di sviluppo distribuito influenza e cambia gli approcci della programmazione.

Contents

1 Obiettivi

1.1 Definizione degli obbietivi

L'obiettivo principale del progetto è quello di studiare e analizzare i concetti principali che si celano dietro il paradigma Serverless e di capirne e provarne ad utilizzare le architetture tipiche analizzandone le caratteristiche che hanno fatto si che prendesse così tanto piede nello scenario tecnologico odierno. I punti su cui focalizzerò il mio studio saranno:

- Architetture e concetti su cui si basa il paradigma Serverless;
- Vantaggi e svantaggi dell' approccio Serverless;
- Analisi dei principali servizi AWS quali Cognito, Lambda, DynamoDB, S3 e delle loro applicazioni all'interno di architetture Baas e Faas;
- Analisi dell'evoluzione di API REST in seguito all'integrazione di architetture Faas e Baas;
- Analisi e confronto con architetture e paradigma Paas.
- Creazione di applicazioni per scenari di test quali: autenticazione(registrazione, login, recupero password), CRUD, storage dati in c# per apprendere meglio i vari concetti.

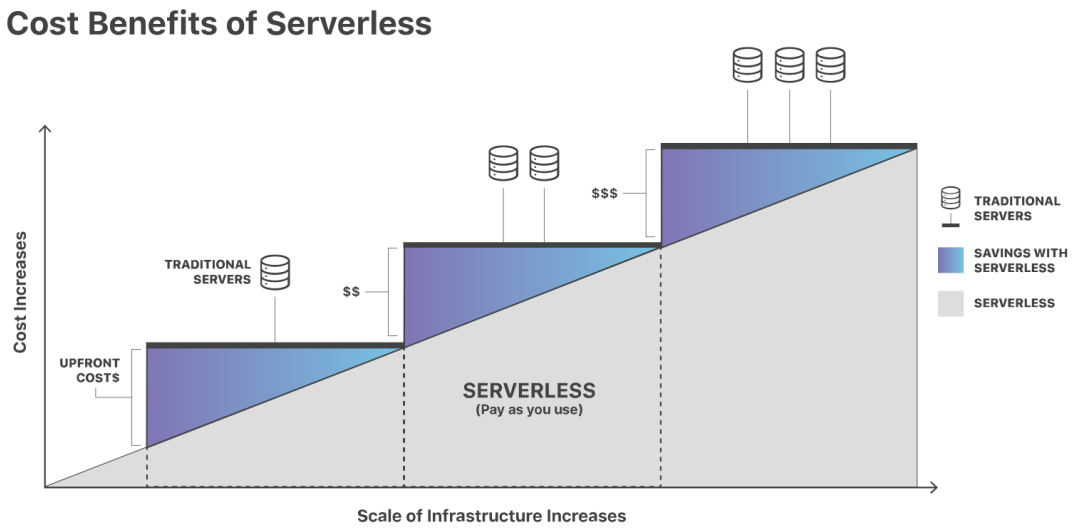
2 Architetture e paradigma Serverless

Nei seguenti capitoli ci concentreremo sul capire cosa significa serverless, quali sono le architetture serverless e perchè è diventato così popolare.

2.1 Serverless Definizione

"What does serverless mean for us?" chiese Chris Munns, senior developer per AWS, durante la conferenza re:Invent 2017 gli venne risposto: "There's no servers to manage or provision at all. This includes nothing that would be bare metal, nothing that's virtual, nothing that's a container, anything that involves you managing a host, patching a host, or dealing with anything on an operating system level, is not something you should have to do in the serverless world." Amazon utilizza i termini "serverless" e "FaaS" in modo intercambiabile e, per chi operana nel mondo AWS, è corretto. Ma nel mondo più ampio dello sviluppo del software, non sono sinonimi. I framework serverless possono, e recentemente sempre più, superare i confini dei fornitori di servizi Faas. L'ideale è che se davvero non ti interessa chi o cosa fornisce il servizio, allora non dovresti essere vincolato dalle regole e dalle restrizioni del fornitore cloud. Nel libro Designing Distributed Systems

, Brendan Burns, Microsoft Distinguished Engineer and Kubernetes co-creator di Kubernetes, avverte i lettori di non confondere il serverless con Faas. Sebbene sia vero che le implementazioni FaaS oscurano l'identità e la configurazione del server host al client, ciò non solo è possibile ma, in determinate circostanze, è desiderabile per un'organizzazione eseguire un servizio FaaS su server che non solo gestisce esplicitamente. FaaS può sembrare serverless da un punto di vista, ma egli come altri sostenitori pensa che un modello di programmazione veramente serverless corrisponde a un modello di distribuzione serverless, non vincolato ad un singolo server o a un singolo fornitore di servizi."The idea is, it's serverless. But you can't define something by saying what it's not," disse David Schmitz, a developer for Germany-based IT consulting firm Senacor Technologies, in una conferenza a Zurigo e proseguì citando la definizione di serverless presa dal sito Web di AWS "Dicono che puoi fare le cose senza pensare ai server. I server Esistono, ma non ci pensi; non è necessario averli e configurarli manualmente, ridimensionarli, gestirli e ripararli. Puoi concentrarti su qualsiasi cosa tu stia realmente facendo, ciò significa che il punto di forza è che puoi concentrarti su ciò che conta e puoi ignorare tutto il resto. Riprese poi dicendo che ci si accorgerà che è una bugia. Insomma, non vi è una vera e propria definizione di serverless, ma possiamo affermare che l'elaborazione serverless è un metodo per fornire servizi di backend in base all'utilizzo. Un provider serverless consente agli utenti di scrivere e distribuire codice senza il fastidio di preoccuparsi dell'infrastruttura sottostante e il servizio scala automaticamente. Il termine serverless è fuorviante perché sono comunque utilizzati dei server in cloud, ma questi vengono astratti ai programmatori, che non devono più occuparsi di mantenimento o configurazione. La spiegazione del successo di questa tecnologia è mostrata nel seguente grafico.



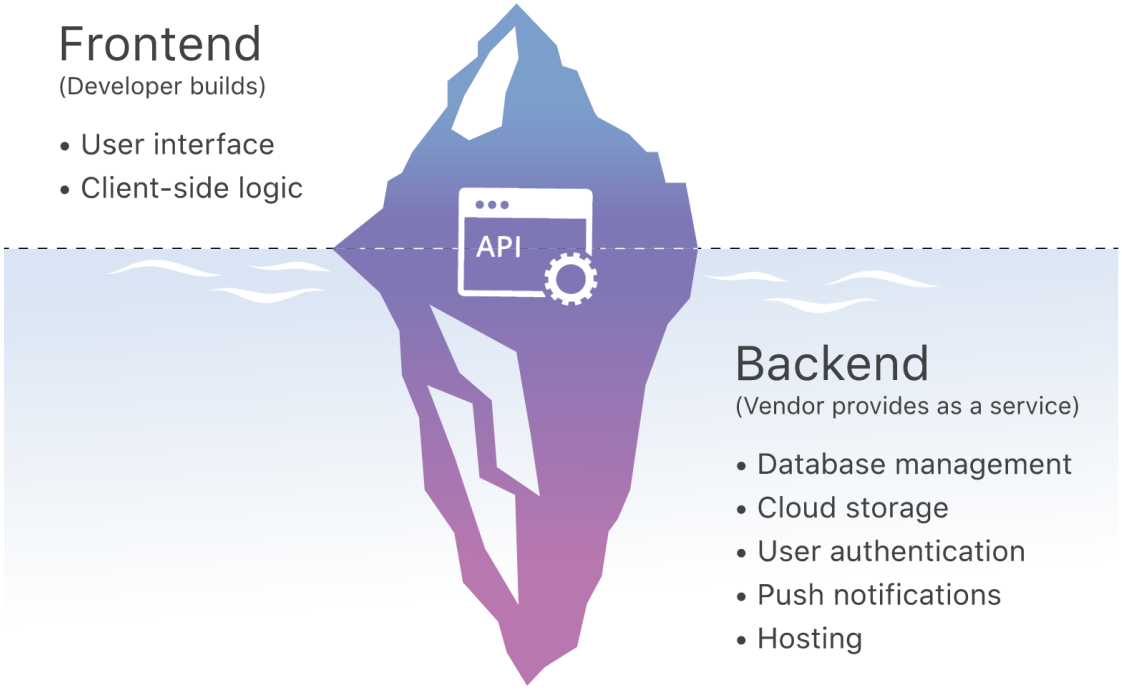
Dal grafico è evidente che il costo iniziale è maggiore, ma se paragonato al costo dell'allestimento di un infrastruttura server fisica interna o in cloud è inferiore e questo trend rimane anche al crescere della dimensione dell'infrastruttura. La differenza sta nel pagare le risorse in base all'elaborazione.

2.2 Architetture Serverless

Ora che abbiamo chiarito il concetto di Serverless possiamo analizzare meglio le tipiche architetture che supportano e compongono questo paradigma e che ne permettono il funzionamento, generando quel livello in più di astrazione. Ricordiamo che le architetture serverless derivano dall'uso di architetture FaaS e BaaS, come mostrato nell'immagine seguente.

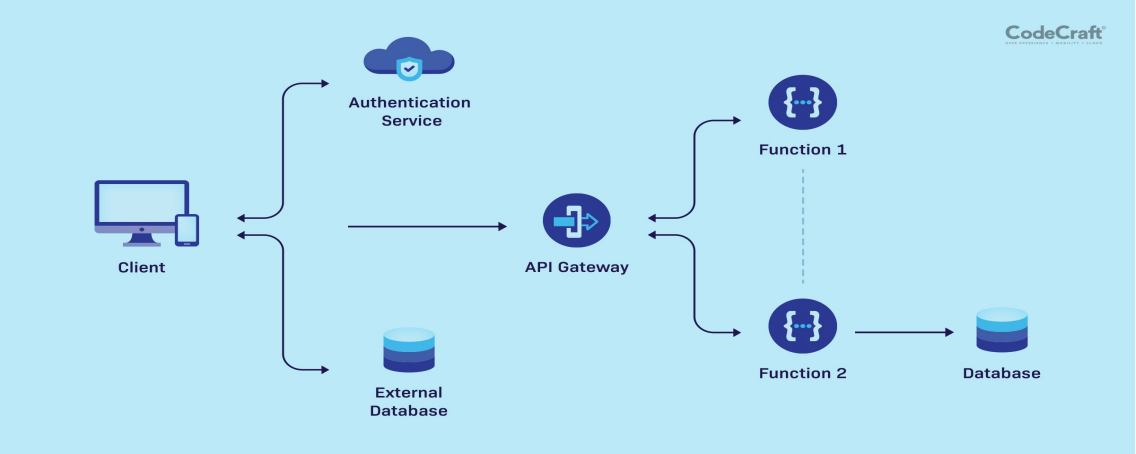
2.3 Backend as a Service

L'immagine seguente ci da un'idea dello scenario tipico di applicazioni web e mobile e ci permette di capire quali sono le principali attività che un svolge un Beckend.



Applicazioni web e mobile richiedono un analogo insieme di funzionalità sul backend come: autenticazione, notifiche push, l'integrazione con le reti social e lo storage di dati. Ognuno di questi servizi ha la propria API che deve essere incorporate singolarmente in una app, un processo che può richiedere molto tempo per gli sviluppatori di applicazioni. I provider di BaaS formano un ponte tra il frontend di un'applicazione e vari cloud-based backend tramite una API unificata e SDK (pacchetto di sviluppo per applicazioni). Fornire un metodo costante e coerente per gestire i dati di backend significa consente agli sviluppatori di non dover sviluppare il proprio backend per ciascuno dei servizi che le loro applicazioni hanno bisogno di accedere, potenzialmente integrando i servizi di BaaS risparmiano tempo e denaro. Anche se è simile ad altri strumenti di sviluppo di cloud computing, come 'software as a service' (SaaS),' Infrastructure as a Service' (IaaS) e 'Platform as a Service' (PaaS), BaaS si distingue da questi altri servizi poiché riguarda in particolare le esigenze del cloud computing di sviluppatori di applicazioni web e mobile, fornendo uno strumento unificato per collegare le loro applicazioni ai servizi cloud. Il BaaS consente quindi di effettuare un "outsourcing" di gran parte degli aspetti

backend di un'applicazione web o mobile. Essendo "plug-and-play", l'integrazione di tali servizi nei sistemi di un'azienda è estremamente più semplice e, talvolta, più affidabile, rispetto a svilupparli internamente. Lo schema seguente riassume un architettura BaaS dove il client si interfaccia con un sistema di autenticazione, ottiene un token e con quello può accedere a delle funzionalità specifiche di back-end specifiche dell'applicazione o ai dati gestiti nei database esterni. Ad oggi,



l'architettura BaaS viene utilizzata soprattuto dalle applicazioni mobile, si parla quindi di Mobile Backend as a Service, MBaas. I vantaggi di BaaS sono i seguenti:

- Riduzione dei costi hardware
- Riduzione dei costi di sviluppo
- Riduzione del time to market
- Alta scalabilità
- Esternalizzazione delle responsabilità
- Focus su UI/UX del frontend

2.4 Functions as a Service

Function-as-a-Service, o FaaS, è un modello di cloud computing basato su eventi, che viene eseguito in container stateless; le funzioni gestiscono la logica e lo stato lato server utilizzando dei servizi. Questo modello consente agli sviluppatori di creare, eseguire e gestire i pacchetti applicativi come funzioni, senza doversi occupare della propria infrastruttura. FaaS è una modalità di esecuzione dell'elaborazione serverless, con la quale gli sviluppatori scrivono la logica delle loro applicazioni che viene eseguita in container totalmente gestiti da una piattaforma. In genere questa piattaforma si trova nel cloud, ma il modello si sta espandendo per

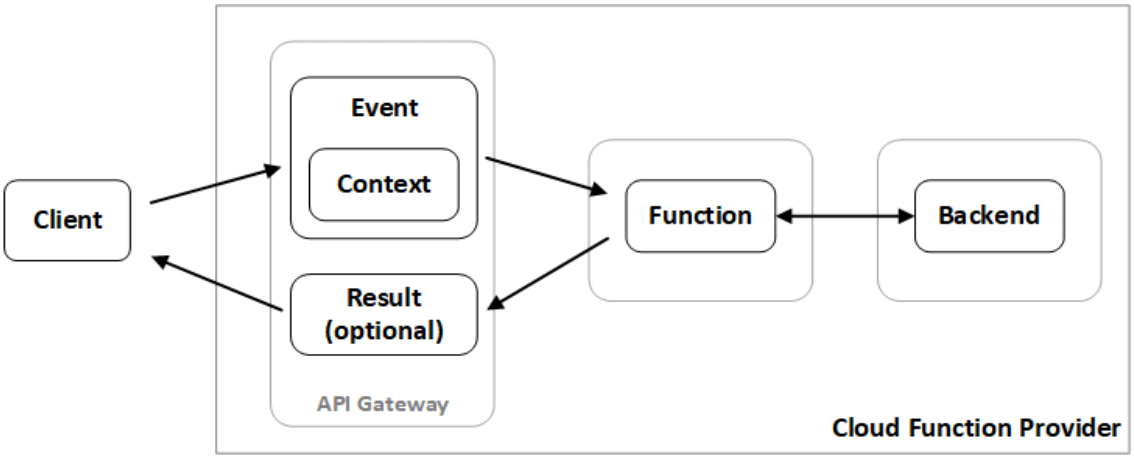
integrare deployment on premise e ibridi. Nel modello serverless, gli sviluppatori non si occupano dei problemi legati all'infrastruttura, come la gestione o il provisioning dei server o l'allocazione delle risorse. Una funzione è un elemento software che esegue la logica dell'applicazione. Le applicazioni sono costituite da numerose funzioni, il nuovo compito dello sviluppatore è quello di scorporarle dando vita ad una architettura a microservizi dove queste funzioni dialogano o svolgono compiti specifici dell'ecosistema. L'impiego di un modello FaaS è una delle modalità per realizzare un'app con un'architettura serverless. Alcuni esempi di FaaS molto diffusi sono:

- AWS Lambda
- Google Cloud Functions
- Microsoft Azure Functions (open source)
- OpenFaaS (open source)

L'infrastruttura FaaS viene utilizzata on demand, principalmente mediante un modello di esecuzione basato su eventi; ciò crea un livello superiore di astrazione che permette di eseguire le applicazioni in risposta a eventi; inoltre in questo modo l'infrastruttura è presente solo quando necessario, senza richiedere l'esecuzione costante dei processi server in background, come accade con il modello PaaS Platform as a Service. Le moderne soluzioni PaaS offrono capacità serverless integrate nei comuni flussi di lavoro utilizzati dagli sviluppatori per distribuire le applicazioni, rendendo meno evidenti i confini tra PaaS e FaaS. Nella realtà, le applicazioni moderne sono composte da una combinazione di funzioni, microservizi e servizi a lunga esecuzione. Un provider cloud rende la funzione disponibile e gestisce l'allocazione delle risorse. Essendo basate sugli eventi e non sulle risorse, le funzioni sono facilmente scalabili. Esistono vincoli architettonici come ad esempio limiti di tempo per l'esecuzione di una determinata funzione; per questa ragione, una funzione deve poter essere avviata ed eseguita rapidamente. Le funzioni si avviano in millisecondi ed elaborano le singole richieste. In presenza di numerose richieste simultanee, il sistema crea il numero di copie della funzione necessarie per soddisfare le domande. Al diminuire delle richieste, l'applicazione procede automaticamente con l'eliminazione delle copie non necessarie. La scalabilità dinamica è un vantaggio del modello FaaS; è anche conveniente, poiché i provider addebitano solo il costo delle risorse utilizzate e non dei tempi morti.Un servizio basato su eventi che richiede la scalabilità orizzontale può funzionare bene sia come funzione sia come applicazione RESTful. Il modello FaaS è perfetto per transazioni con volumi elevati, carichi di lavoro con frequenza sporadica come la generazione di report, l'elaborazione di immagini o attività programmate. Esempi di utilizzo comuni sono l'elaborazione dei dati, i servizi IoT, le app mobile o web. I vantaggi di FaaS:

- Maggiore produttività degli sviluppatori e tempi di sviluppo più rapidi
- Nessuna responsabilità per la gestione dei server
- Scalabilità facilitata, con estensione orizzontale gestita dalla piattaforma
- Vengono addebitate solo le risorse consumate
- È possibile scrivere le funzioni utilizzando qualsiasi linguaggio di programmazione

L'architettura Faas può essere riassunta come mostrato nell'immagine seguente.



Il client scatena un evento che induce l'attivazione di una funzione, la funzione compone il backend dell'applicazione e può interfacciarsi con esso, ad esempio con le basi dati in esso presenti. La funzione può anche solamente svolgere un operazione di calcolo ad esempio e ritornare un risultato. Non sempre è presente un risultato che viene inviato al client come risposta.

2.5 Vantaggi e svantaggi

Abbiamo già delineato i vantaggi che generano le architetture FaaS e quelle BaaS possiamo riassumerli nei seguenti punti:

- Scalabilità e gestione delle risorse necessarie da parte del provider;
- Fornitura rapida delle risorse in tempo reale, anche con carichi di picco imprevisti e crescita sproporzionata;
- I costi sono calcolati esclusivamente per le risorse utilizzate;
- Elevata tolleranza ai guasti grazie all'infrastruttura hardware flessibile nei data center del provider;

- Eliminazione della fase di configurazione delle macchine di sviluppo/test/produzione;
- Semplificazione dei processi di sviluppo di applicazione client-server, lasciando sviluppare solo le parti specifiche di ogni applicazione e "esternalizzando" le parti comuni;
- Minore tempo e costo di distribuzione e di avvio dei lavori;
- I fornitori del cloud forniscono anche i sistemi di registrazione e monitoraggio per i consumatori;

Purtroppo non ci sono solo vantaggi, le serverless application hanno i seguenti svantaggi:

- L'accesso alle macchine virtuali, al sistema operativo o agli ambienti di runtime resta vietato;
- L'implementazione di strutture serverless è molto impegnativa;
- Elevata dipendenza dal provider ("effetto lock-in") - quando si cambia provider, ad esempio, la maggior parte delle funzioni basate sugli eventi deve essere riscritta;
- Processo di monitoraggio e debug relativamente complicato, poiché non sempre sono possibili analisi approfondite delle prestazioni e degli errori;

E' evidente che questo nuovo paradigma di sviluppo ha rivoluzionato i metodi canonici divenendo uno dei paradigmi odierni più utilizzati. Dagli svantaggi emerge che il segreto sta nell'affidarsi al cloud provider migliore a seconda delle nostre esigenze.

3 Architetture a confronto

Storicamente l'archietettura più simile al modello serverless è quella con Trusted Platform Module. Se guardiamo indietro nel tempo, vedremo che lavorare con il serverless ha molte somiglianze con il modo in cui il team di sviluppatori lavora con i mainframe. C'era un nome di tecnologia come TPM (Transaction Process Monitoring), ad esempio, CICS (Customer Information Control System) di IBM e tutto ciò che faceva era gestire elementi non funzionali. TPM gestisce i problemi di gestione del carico, sicurezza, monitoraggio, distribuzione, ecc. Quindi, TPM può essere definito come un antenato del Serverless.

3.1 Confronto con Infrastructure as a Service(IaaS)

IaaS offre un'infrastruttura completa, come richiesto, e gli sviluppatori devono prima installare una macchina virtuale, configurarla e poi distribuire l'app. IaaS viene eseguito con la virtualizzazione dell'infrastruttura richiesta. Tutto deve essere gestito dal team dello sviluppatore.

3.2 Confronto con Container as a Service(CaaS)

CaaS viene utilizzato dove vogliamo la massima flessibilità. Perché offre un ambiente completamente controllato in cui gli sviluppatori possono distribuire l'app, come vogliono, e possono eseguire la loro app con pochissime o quasi nessuna dipendenza dal lato client. Gli sviluppatori però devono gestire il controllo del traffico.

3.3 Confronto con Platform as a Service(PaaS)

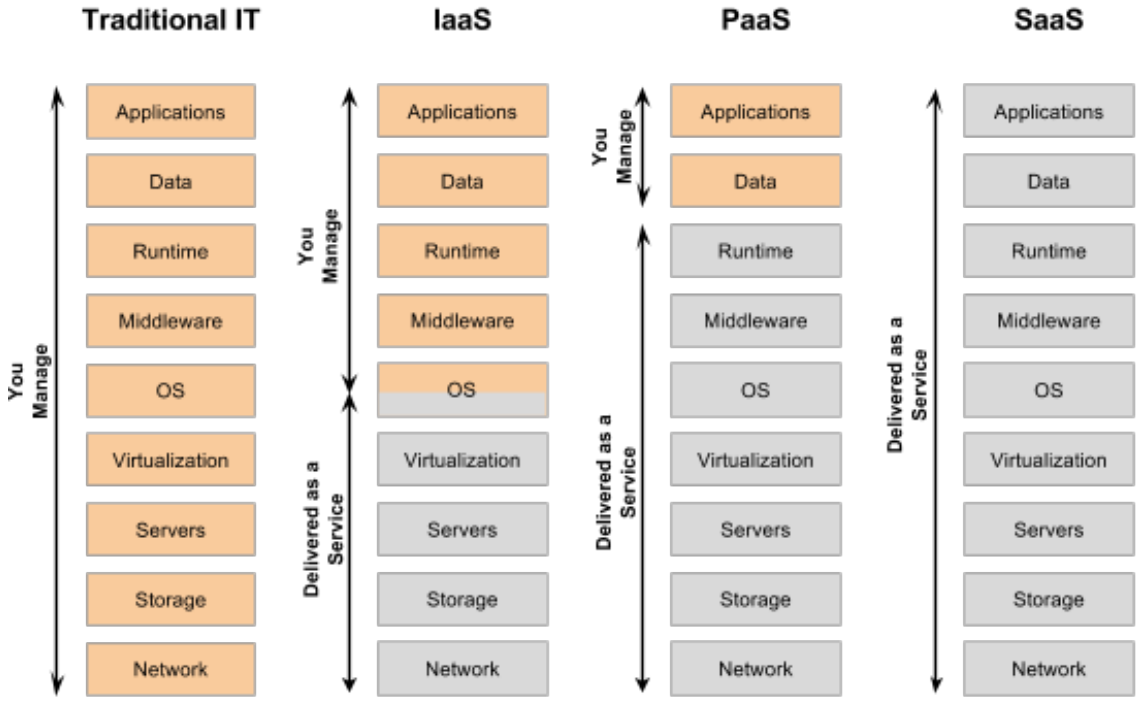
Poi è arrivato il momento di concentrarsi su PaaS, che è stato un passo verso il Serverless, è la versione migliorata di CaaS, in cui lo sviluppatore non deve preoccuparsi del sistema operativo e del suo funzionamento. Lo sviluppare si limita a distribuire l'app o il servizio sulla piattaforma.La differenza sta nel fatto che in Serverless il ridimensionamento avviene istantaneamente senza alcuna pianificazione preliminare, è automatico, nel caso di PaaS, il ridimensionamento può essere fatto ma non è automatico sono gli sviluppatori a doversi occupare di gestirlo, va quindi pre-configurato. L'architettura FaaS è quella più simile all'architettura PaaS, oltre alla scalabilità autogestita offre una migliore disponibilità e un ridotto effetto lock-in nei confronti del provider cloud. Inoltre il Serverless è microgestito, quindi le risorse amministrative interne possono essere utilizzate per altre attività.

3.4 Serverless Vs. Containers

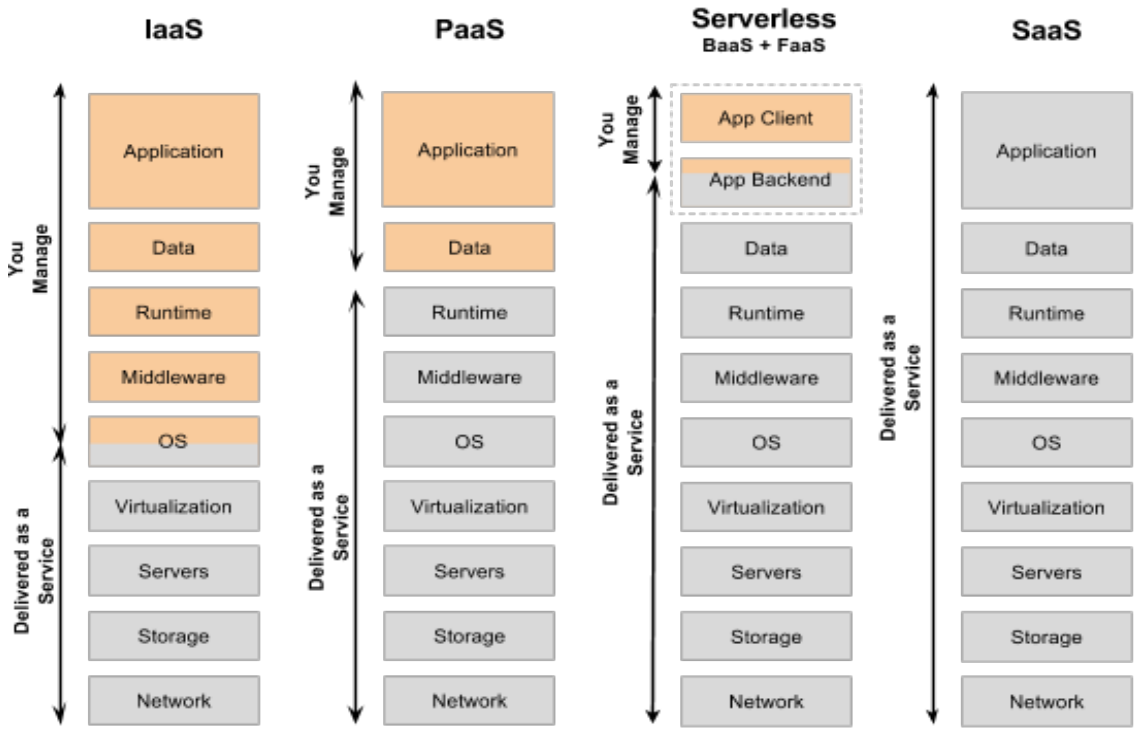
I container lavorano su un'unica macchina, alla quale sono assegnati inizialmente; possono, però, essere riposizionati. L'assegnazione del server in serverless è responsabilità del provider, e viene attaccata quando un evento attiva la funzione. In serverless, non è confermato che lo stesso scope verrà eseguito più volte sulla stessa macchina, anche se proviamo due volte consecutive. Nei container il ridimensionamento deve essere fatto dagli sviluppatori, non è facile da gestire, ma può essere fatto. In serverless il ridimensionamento è automatico. Nei container le tariffe sono fisse, perché essi devono funzionare continuamente, che ci sia o meno traffico. In caso di serverless dobbiamo pagare in base a quanto usiamo, che viene valutato dal tempo di esecuzione della nostra funzione. Nel caso dei container, la responsabilità della gestione è nelle mani dei cloud consumers. Mentre nel caso del serverless, gli sviluppatori devono inviare la propria logica, o parte di essa, come funzione; il cloud provider gestirà tutto il resto.

3.5 Architetture a confronto

E' interessante vedere di cosa bisogna occuparsi a seconda dell'architettura che si sceglie di utilizzare, questo influisce sia sullo sviluppo che sul mantenimento dell'applicazione e dell'architettura che la supporta. Il grafico seguente ci mostra ciò di cui dobbiamo occuparci a seconda del paradigma architettura che stiamo utilizzando, nelle architetture classiche.



E' evidente che l'architettura tradizionale ad oggi obsoleta. Risulta interessante però vedere dove si posizionano le architetture Serverless in questo grafico.



E' più che evidente che adottare il paradigma serverless semplifica di molto i processi di produzione e mantenimento di una applicazione. A questo corrisponde un aumento di complessità in fase di progettazione dell'applicazione che approfondiremo in seguito.

4 Principali servizi AWS

4.1 Lambda

4.2 Cognito

4.3 Dynamo DB

4.4 S3

5 Analisi dell'evoluzione agli approcci di progettazione di API REST Application

Con l'evolversi delle architetture e dei paradigmi di sviluppo è normale che cambino anche gli approcci alla progettazione. In questa sezione si vuole evidenziare dove si sposta il focus dell'attenzione durante la progettazione di API REST.

5.1 Evoluzione della progettazione degli ambienti

In un approccio tradizione parte del focus sarebbe speso nell'individuazione delle caratteristiche fisiche della macchina su cui effetture il deploy; subito dopo si passerebbe alla scelta del sistema operativo; infine ci sarebbe una fase di configurazione di tutta la macchina al fine di renderla pronta ad accogliere e mettere in funzione la nostra applicazione. In un approccio non tradizione potremmo replicare quelle impostazioni o direttamente l'istanza della macchina e creare i tre tipici ambienti che vengono utilizzati a livello aziendale: sviluppo, test, produzione. Sembra banale ma non lo è, qui si aprono tantissimi scenari reali a difficoltà variabile a seconda dei vincoli che il cliente pone; parte del focus è dedita anche alla stima dei carichi di lavoro, elemento su cui si basa il primo dei tre passaggi definiti precedentemente. L'approccio Serverless taglia completamente questa pratica, la quantità di risorse messe a disposizione, il loro sistema operativo e la loro configurazione viene completamente gestita dal cloud provider. Per la creazione degli ambienti di sviluppo, test e produzione è sufficiente configurare ed utilizzare gli Environment nel template e fare deploy; a seconda del tipo di applicazione, l'ambiente di sviluppo potrebbe essere quello in esecuzione sulla macchina locale.

5.2 Evoluzione delle logiche di progettazione

Nell'approccio classico la parte progettuale prevederebbe l'individuazione delle logiche di funzionamento e di gestione dei dati, tutto in un'unica applicazione. Seguendo uno sviluppo più attento al devOps, potremmo: scindere delle logiche di funzionamento; containerizzarle e cercare di creare una struttura a micro-servizi. In un approccio di questo tipo c'è ancora una parte Ops realativa alla configurazione dei container. Nell'approccio Serveless possiamo occuparci unicamente di pensare a come dar vita ad un applicazione a micro-servizi, suddividendo le logiche di funzionamento avendo garantita alta disponibilità e alta scalabilità, ci si concentra solo sugli aspetti dev. L'integrazione di servizi BaaS presenti in una applicazione serverless aiuta e favorisce la suddivisione dell'applicazione in micro-servizi con la divisione di carichi e compiti, favorendo modularità e riusabilità.

5.3 Evoluzione del Deploy

Nei modelli tradizionali il deploy è sempre complesso, si accede alla macchina o in RDP(Remote Desktop Protocol) o in SSH(Secure SHell) e si va a pubblicare la nostra applicazione, si crea una copia di backup della build precedente e si manda in esecuzione la nuova build. Nell'apporccio Serveless il deploy è supportato e monitorato dal cloud provider e si occuperà lui di fare restore nel caso in cui qualcosa vada storto, inoltre non c'è interruzione nel servizio. La semplificazione fornita porta a distogliere il focus da questa fase.

//spostare prima della presentazione dei progetti.

6 Requisiti

6.1 Requisiti funzionali

Per i requisiti funzionali variano a seconda degli esempi pratici che verranno prodotti. Iniziamo a fare una distinzione dei requisiti funzionali in un'architettura Backend as a Service e in un'architettura Functions as a Service che descriveremo nello specifico in seguito. In generale per la produzione di parte di questi progetti è necessario essere in possesso di credenziali AWS e di un ruolo con i permessi di accesso ai vari servizi.

6.2 Requisiti funzionali Baas

Il Backend as a Service è un modello di calcolo basato su cloud, che automatizza e gestisce il lato back-end dello sviluppo di un'applicazione web o mobile. Ha come scopo principale quello di aiutare gli sviluppatori con l' autenticazione utente, l' archiviazione cloud o hosting di dati e file. Implementeremo un meccanismo di autenticazione che permette al client di autenticarsi senza dover passare per il server. Si fa presente che l'approccio ha la finalità di scorporare la gestione di questa fase, presente in moltissime applicazioni, dal server, al fine di rendere mantenibile solo il frontend. E' necessario prevedere la presenza di Trigger per poter far si che ad azione corrisponda reazione.

6.3 Requisiti funzionali Faas

Il Functions as a Service permette di eseguire i pezzi modulari del codice, tipicamente prevede un uso integrato, è dunque necessario definire ed individuare quali saranno le funzionalità che devono garantire scalabilità, vedremo un'applicazione di questo approccio legato alle API Rest; uno scenario tipico dove alcune chimate sono più costose di altre in termini di risorse, scorporare l'esecuzione di alcune chiamate API dal server ci permette di migliorare le prestazioni e garantire disponibilità e scalabilità. Vedremo anche un uso di Faas come Baas, che ad oggi è la pratica più diffusa e permette di gestire intere applicazioni secondo una logica a micro-servizi.

6.4 Requisiti non funzionali

Per poter comprendere a pieno le potenzialità di questo nuovo paradigma sarebbe necessario conoscere i processi e le metodologie di sviluppo di una classica applicazione client-server. Gli esempi presentati nel progetto, mirano a garantire buona scalabilità e facilità d'uso sia per l'utente che per lo sviluppatore; inoltre il sistema deve garantire disponibilità e consistenza all'interno dell'architettura e permettere un approccio a micro-servizi.

6.5 Scenarios

Informal description of the ways users are expected to interact with your project. It should describe *how* and *why* a user should use / interact with the system.

6.6 Self-assessment policy

- How should the *quality* of the *produced software* be assessed?
- How should the *effectiveness* of the project outcomes be assessed?

7 Requirements Analysis

Is there any implicit requirement hidden within this project's requirements? Is there any implicit hypothesis hidden within this project's requirements? Are there any non-functional requirements implied by this project's requirements?

What model / paradigm / techonology is the best suited to face this project's requirements? What's the abstraction gap among the available models / paradigms / techonologies and the problem to be solved?

8 Design

This is where the logical / abstract contribution of the project is presented.

Notice that, when describing a software project, three dimensions need to be taken into account: structure, behaviour, and interaction.

Always remember to report **why** a particular design has been chosen. Reporting wrong design choices which has been evalued during the design phase is welcome too.

8.1 Structure

Which entities need to by modelled to solve the problem? (UML Class diagram)

How should entities be modularised? (UML Component / Package / Deployment Diagrams)

8.2 Behaviour

How should each entity behave? (UML State diagram or Activity Diagram)

8.3 Interaction

How should entities interact with each other? (UML Sequence Diagram)

9 Implementation Details

Just report interesting / non-trivial / non-obvious implementation details.

This section is expected to be short in case some documentation (e.g. Javadoc or Swagger Spec) has been produced for the software artefacts. This this case, the produced documentation should be referenced here.

10 Self-assessment / Validation

Choose a criterion for the evaluation of the produced software and **its compliance to the requirements above**.

Pseudo-formal or formal criteria are preferred.

In case of a test-driven development, describe tests here and possibly report the amount of passing tests, the total amount of tests and, possibly, the test coverage.

11 Deployment Instructions

Explain here how to install and launch the produced software artefacts. Assume the softaware must be installed on a totally virgin environment. So, report **any** configuration step.

Gradle and Docker may be useful here to ensure the deployment and launch processes to be easy.

12 Usage Examples

Show how to use the produced software artefacts.

Ideally, there should be at least one example for each scenario proposed above.

13 Conclusions

Recap what you did

13.1 Future Works

Recap what you did *not*

13.2 What did we learned

Racap what did you learned

Stylistic Notes

Use a uniform style, especially when writing formal stuff: X, X, $\mathbf{X}$, $\mathcal{X}$, `X` are all different symbols possibly referring to different entities.

This is a very short paragraph.

This is a longer paragraph (notice the blank line in the code). It composed by several sentences. You're invited to use comments within `.tex` source files to separate sentences composing the same paragraph.

Paragraph should be logically atomic: a subordinate sentence from one paragraph should always refer to another sentence from within the same paragraph.

The first line of a paragraph is usually indented. This is intended: it is the way LaTeX lets the reader know a new paragraph is beginning.

Use the `listing` package for inserting scripts into the LaTeX source.