

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 miliardi 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 incorporare 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 includendo 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	4
1.1	Definizione degli obiettivi	4
2	Architetture e paradigma Serverless	4
2.1	Serverless Definizione	4
2.2	Architetture Serverless	6
2.3	Backend as a Service	7
2.4	Functions as a Service	8
2.5	Vantaggi e svantaggi	10
3	Architetture a confronto	11
3.1	Confronto con Infrastructure as a Service(IaaS)	12
3.2	Confronto con Container as a Service(CaaS)	12
3.3	Confronto con Platform as a Service(PaaS)	12
3.4	Serverless Vs. Containers	12
3.5	Architetture a confronto	13
4	Analisi dell'evoluzione agli approcci di progettazione di API REST Application	14
4.1	Evoluzione della progettazione degli ambienti	14
4.2	Evoluzione delle logiche di progettazione	15
4.3	Evoluzione del Deploy	15
5	Principali servizi AWS	15
5.1	Lambda	16
5.1.1	Lambda function	17
5.1.2	Lambda Application	17
5.2	Cognito	18
5.2.1	User Pool	19
5.3	Identity Pool	19
5.4	Dynamo DB	19
5.5	S3	20
5.5.1	Bucket	21
5.5.2	Oggetti	21
5.5.3	Chiavi	21
6	Sviluppo di un'applicazione Serverless: AWSServerlessApplication	22
7	Requisiti	22
7.1	Requisiti funzionali	22
7.2	Requisiti non funzionali	23
7.3	Scenarios	23
7.4	Self-assessment policy	23

8	Requirements Analysis	23
9	Design	23
9.1	Structure	23
9.2	Behaviour	24
9.3	Interaction	24
10	Implementation Details	24
11	Self-assessment / Validation	24
12	Deployment Instructions	24
13	Usage Examples	24
14	Conclusions	24
14.1	Future Works	25
14.2	What did we learned	25

1 Obiettivi

1.1 Definizione degli obiettivi

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 sì 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

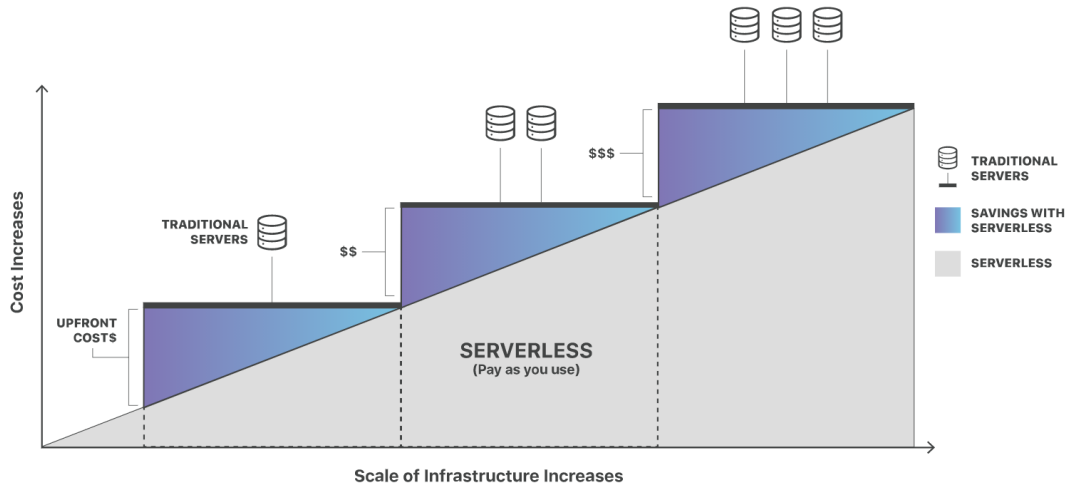
Nei seguenti capitoli ci concentreremo sul capire cosa significa serverless, quali sono le architetture serverless e perchè è diventato così popolare.

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 opera 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.

Cost Benefits of Serverless



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.

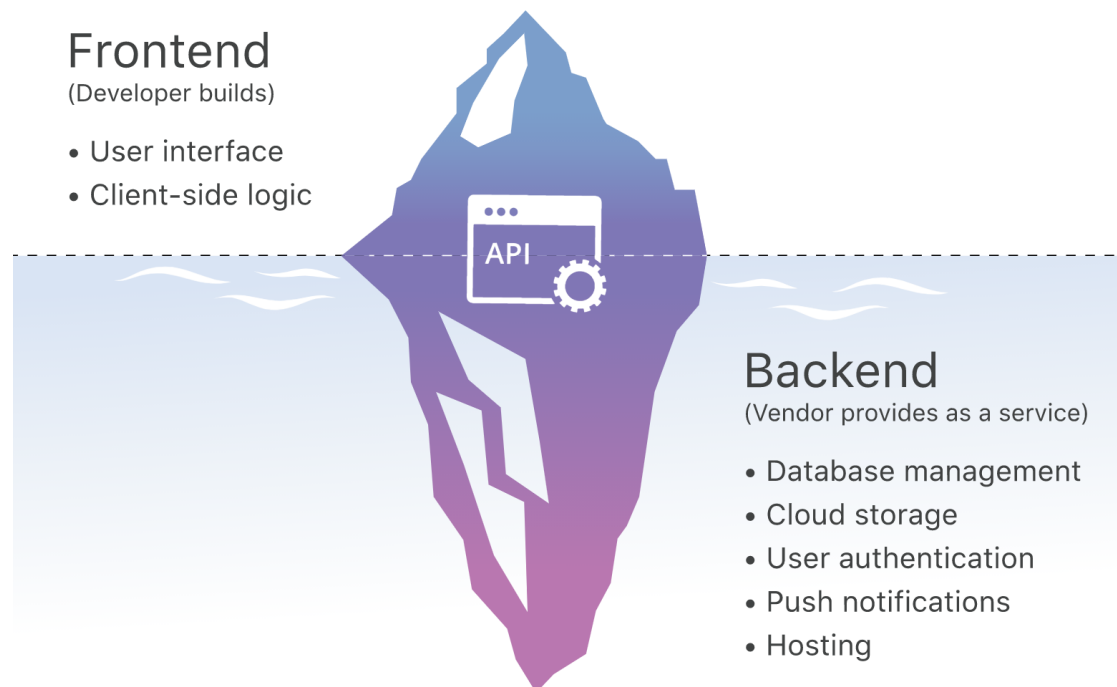
2.2 Architetture Serverless

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.



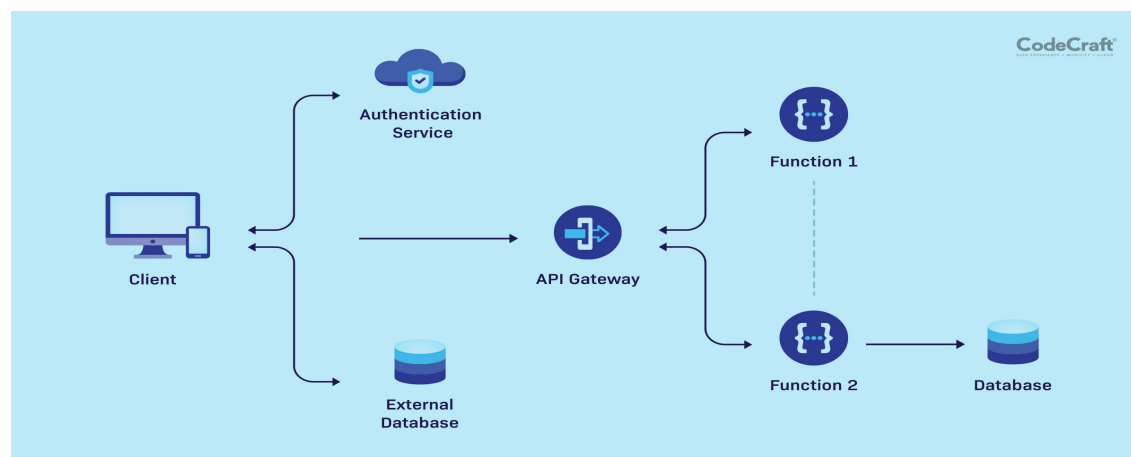
2.3 Backend as a Service

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 Backend.



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 incorporata 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 consentire 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 backend specifiche dell'applicazione o ai dati gestiti nei database esterni. Ad oggi,



l'architettura BaaS viene utilizzata soprattutto 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

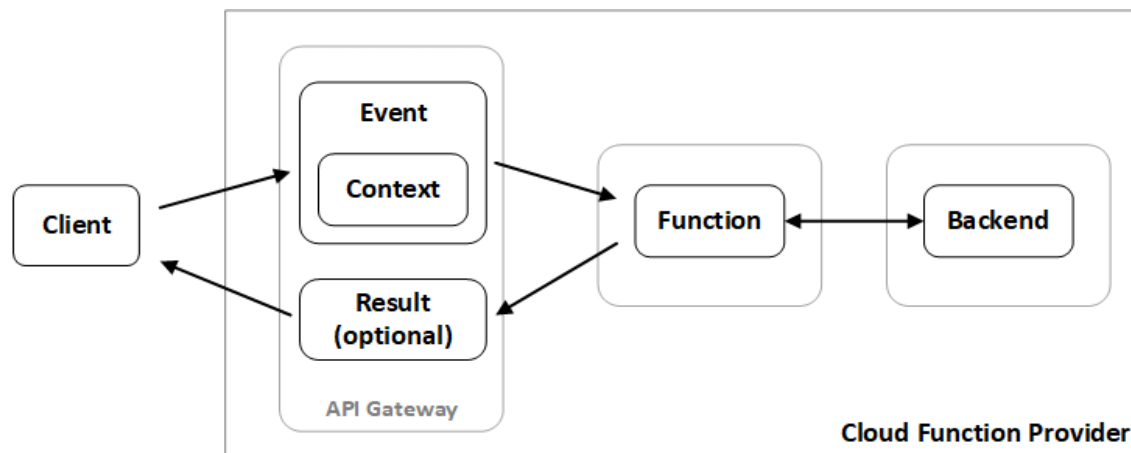
integrare deployment on premise e ibridi. Nel modello serverless, gli sviluppatori non si occupano dei problemi legati all'infrastruttura, come la gestione o il provisioning dei server o l'allocazione delle risorse. Una funzione è un elemento software che esegue la logica dell'applicazione. Le applicazioni 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 architetturici 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'architettura 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)

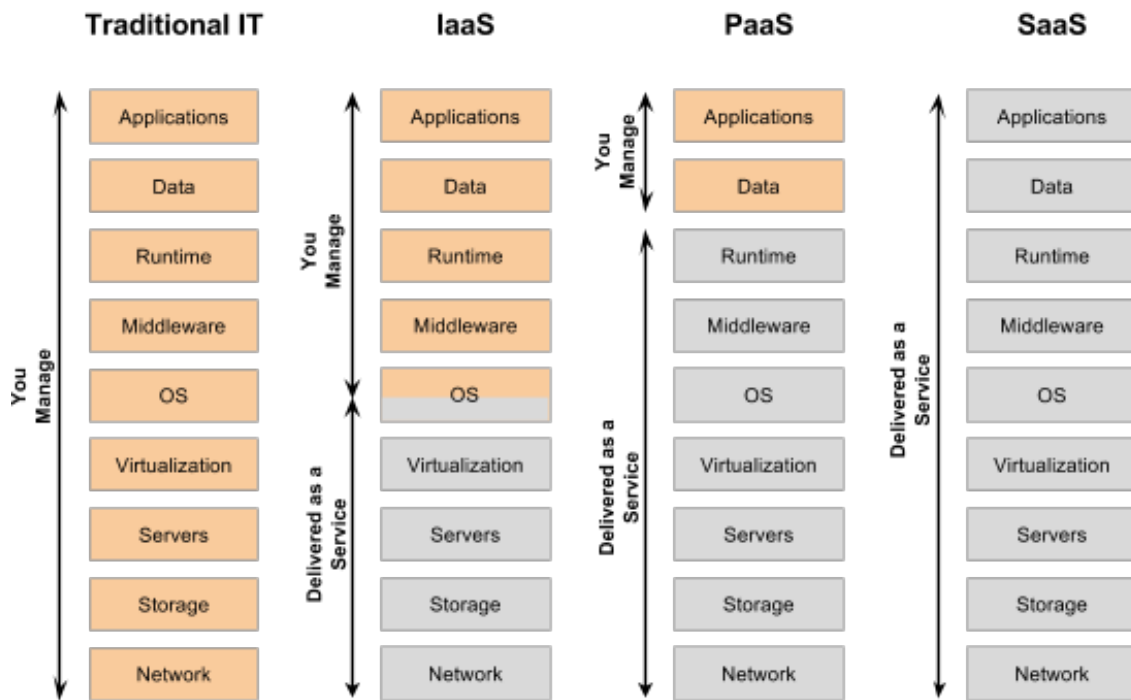
Poi è arrivato il momento di concentrarsi su PaaS, che è stato un passo verso il Serverless, è la versione migliorata di CaaS, in cui lo sviluppatore non deve preoccuparsi del sistema operativo e del suo funzionamento. Lo sviluppare si limita a distribuire l'app o il servizio sulla piattaforma. La differenza 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

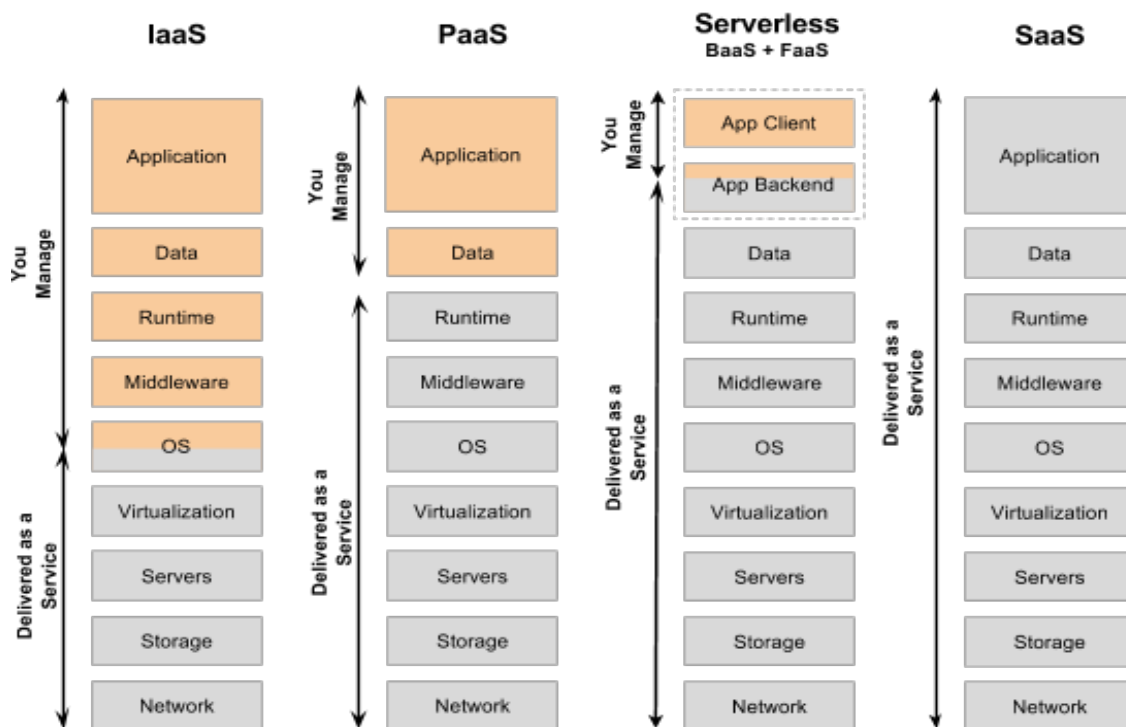
I container lavorano su un'unica macchina, alla quale sono assegnati inizialmente; possono, però, essere riposizionati. L'assegnazione del server in serverless è responsabilità del provider, e viene attaccata quando un evento attiva la funzione. In serverless, non è confermato che lo stesso scope verrà 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 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.

4.1 Evoluzione della progettazione degli ambienti

In un approccio tradizione parte del focus sarebbe speso nell'individuazione delle caratteristiche fisiche della macchina su cui effettuare 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 am-

bienti 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 dedicata 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. L'approccio tradizionale genera un focus anche sulla manutenzione dell'architettura.

4.2 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 relativa alla configurazione dei container. Nell'approccio Serverless 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.

4.3 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'approccio Serverless 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.

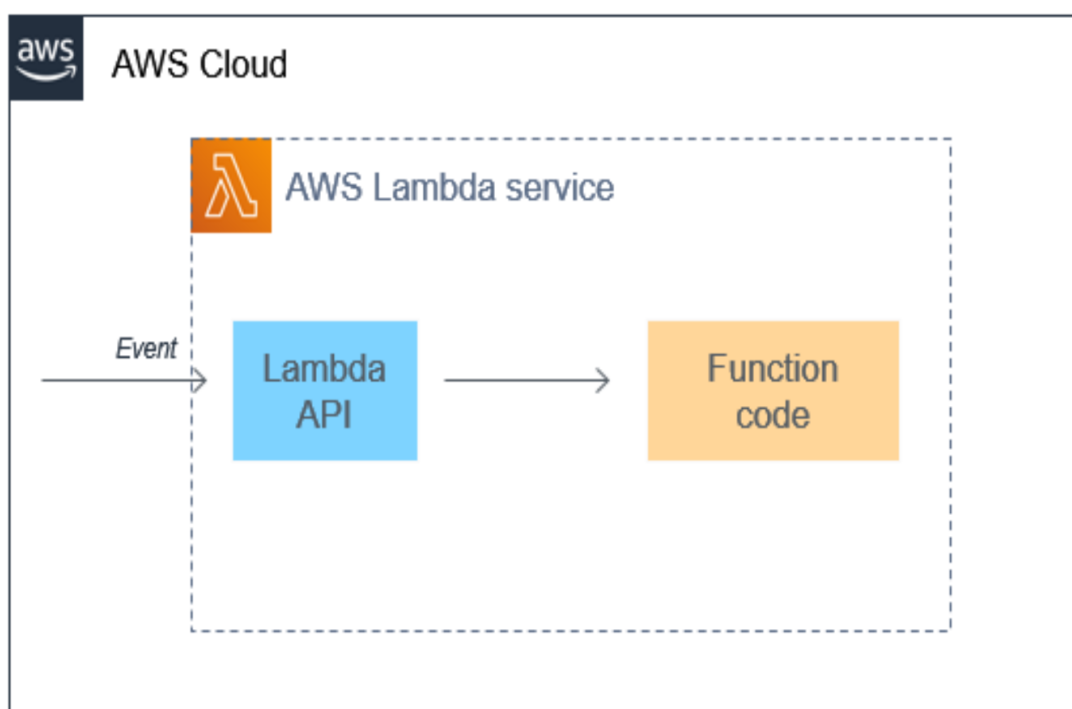
5 Principali servizi AWS

Al fine di comprendere gli esempi proposti in seguito ritengo necessario conoscere le tecnologie che si stanno utilizzando. Amazon Web Services fornisce un livello di astrazione molto elevato, è molto complesso arrivare a capire come funzionano i servizi che mette a disposizione ad un livello di astrazione più basso; non è negli

interessi dello stesso Amazon farci conoscere come fanno i suoi servizi a funzionare, pertanto si fa veramente fatica a trovare qualcosa di certo che ci permetta di capire le architetture interne e fare reverse engineering sui servizi. Per erogare servizi basati su container credo vengano utilizzati Docker e Kubernetes o tecnologie che permettano servizi simili. Concentriamoci su una panoramica generale degli aspetti principali che ogni servizio offre in modo da capire come utilizzarli ed integrarli nei nostri sistemi.

5.1 Lambda

AWS Lambda è un servizio di calcolo che ti consente di eseguire il codice senza eseguire il provisioning o gestire i server. Lambda esegue il codice su un'infrastruttura di elaborazione ad alta disponibilità ed esegue tutta l'amministrazione delle risorse di elaborazione, inclusa la manutenzione del server e del sistema operativo, il provisioning della capacità e la scalabilità automatica, il monitoraggio e la registrazione del codice. Il funzionamento di Lambda si basa sul event-driven paradigm, è dunque un servizio di elaborazione su richiesta che esegue codice personalizzato in risposta agli eventi come mostrato nell'immagine seguente.



A differenza dei server tradizionali, le funzioni Lambda non vengono eseguite costantemente. L'attivazione di una funzione da parte di un evento, viene chiamata invocazione. Le funzioni Lambda sono volutamente limitate a 15 minuti di durata,

ma in media, tra tutti i clienti AWS, la maggior parte delle chiamate dura solo meno di un secondo. In alcune operazioni di calcolo intensive, l'elaborazione di un singolo evento può richiedere diversi minuti, ma nella maggior parte dei casi la durata è breve. Un evento che attiva una funzione Lambda potrebbe essere qualsiasi cosa, da una richiesta HTTP tramite API Gateway, una pianificazione gestita da una regola EventBridge, un evento IOT o un evento S3. Anche la più piccola applicazione basata su Lambda utilizza almeno un evento; inoltre in architetture a micro-servizi è comune avere eventi per gestire il controllo dei flussi. Esistono anche gli Anti-patterns in Lambda-based applications e sono:

- Le applicazioni monolitiche: tipicamente applicazioni migrate da server tradizionali, istanze EC2, in maniera "drag and drop". Spesso, questo si traduce in una singola funzione Lambda che contiene tutta la logica dell'applicazione che viene attivata per tutti gli eventi. Per un'applicazione Web di base, una funzione Lambda monolitica gestirebbe tutte le route API Gateway e si integrerebbe con tutte le risorse necessarie.
- Lambda as orchestrator: Lambda è progettato per essere all'interno di un flusso o un micro-servizio, non dobbiamo utilizzarlo in situazione come la gestione di un pagamento, dove a seconda del circuito c'è un flusso e ogni pagamento può andare o non andare a buon fine.
- Pattern ricorsivi: Evitare situazioni in cui vi è un rimpallo ricorsivo di eventi che vengono invocati a vicenda.
- Lambda functions che chiamano Lambda Functions, se fatto in maniera ricorsiva è doloso, porta ad un aumento dei tempi di latenza. Può essere utile in chiamate sincrone che non generano eventi.

Lambda si distingue in Lambda Function e Lambda Application.

5.1.1 Lambda function

E' quanto abbiamo descritto fin ora, una parte di codice che viene eseguita ogni volta che viene invocata da un evento.

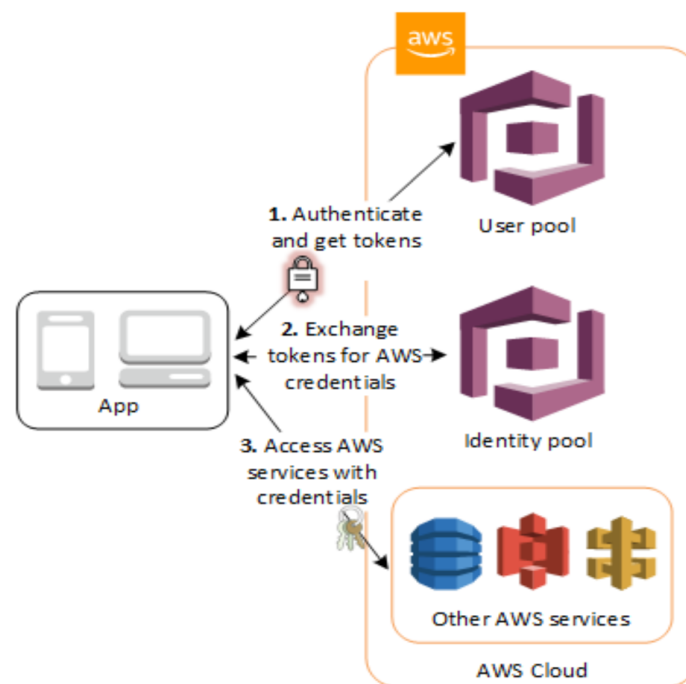
5.1.2 Lambda Application

Le Lambda applications sono la base di un'applicazione Serverless e sono un insieme di funzioni Lambda. Attenzione però è differente creare n funzioni lambda da creare una lambda application. Una lambda application è in grado di auto generare n funzioni suddividendo il tuo codice. Nell'esempio che vedremo in seguito dell'applicazione serverless, è l'API gateway che gestisce le route e invoca delle lambda diverse a seconda di esse, ma la suddivisione del tuo codice in più funzioni viene fatta automaticamente e in maniera ottimizzata da AWS. E' possibile

riprodurre una Lambda application, creando n Lambda functions è configurando a mano l'API Gateway per invocarle a seconda della route scelta. Inoltre vengono definite Lambda application quelle che integrano altri servizi.

5.2 Cognito

Amazon Cognito fornisce autenticazione, autorizzazioni e gestione degli utenti per applicazioni Web e mobile. Gli utenti possono accedere direttamente con un nome utente e una password, oppure tramite terze parti, ad esempio Facebook, Amazon, Google o Apple. I due componenti principali di Amazon Cognito sono i pool di utenti e i pool di identità. I pool di utenti sono directory utente che forniscono opzioni di registrazione e di accesso agli utenti delle tue app. I pool di identità permette di concedere agli utenti l'accesso ad altri servizi AWS. È possibile usare i pool di identità e i pool di utenti separatamente o insieme. Il diagramma mostra uno scenario comune dell'utilizzo di Amazon Cognito.



L'obiettivo è di autenticare l'utente e quindi concedere all'utente l'accesso a un altro servizio. Nella prima fase l'utente si autentica e accede all'applicazione mediante un pool di utenti; riceve il token del pool di utenti; scambia il token del pool di utenti per delle credenziali AWS (che definisco un ruolo) attraverso un pool di identità. Infine, l'utente può utilizzare le credenziali per accedere ad altri servizi AWS come Amazon S3 o DynamoDB.

5.2.1 User Pool

Un pool di utenti è una directory di utenti in Amazon Cognito. Con questo, gli utenti possono accedere all'app Web o mobile tramite Amazon Cognito. Gli utenti, inoltre, possono anche accedere tramite provider di identità social come Google, Facebook, Amazon o Apple e tramite i provider di identità SAML. Sia se gli utenti effettuano l'accesso direttamente o tramite terze parti, tutti i membri del pool di utenti dispongono di un profilo di directory a cui è possibile accedere tramite SDK(Software Development Kit). il pool di utenti fornisce:

- Servizi di registrazione e di accesso.
- Un'interfaccia utente Web integrata personalizzabile per l'accesso degli utenti.
- Accesso social tramite Facebook, Google, Login with Amazon e Accesso con Apple e accesso tramite i provider di identità SAML dal pool di utenti.
- Gestione della directory utente e profili utente.
- Le funzioni di protezione quali l'autenticazione a più fattori (MFA), controllo delle credenziali compromesse, la protezione dal furto dell'account e la verifica dell'e-mail e del numero di telefono.
- Flussi di lavoro personalizzati e migrazione degli utenti tramite i trigger AWS Lambda.
- Una volta completata l'autenticazione di un utente, Amazon Cognito emette dei token Web JSON (JWT) che è possibile utilizzare per proteggere e concedere le autorizzazioni di accesso alle proprie API o per lo scambio di credenziali AWS.

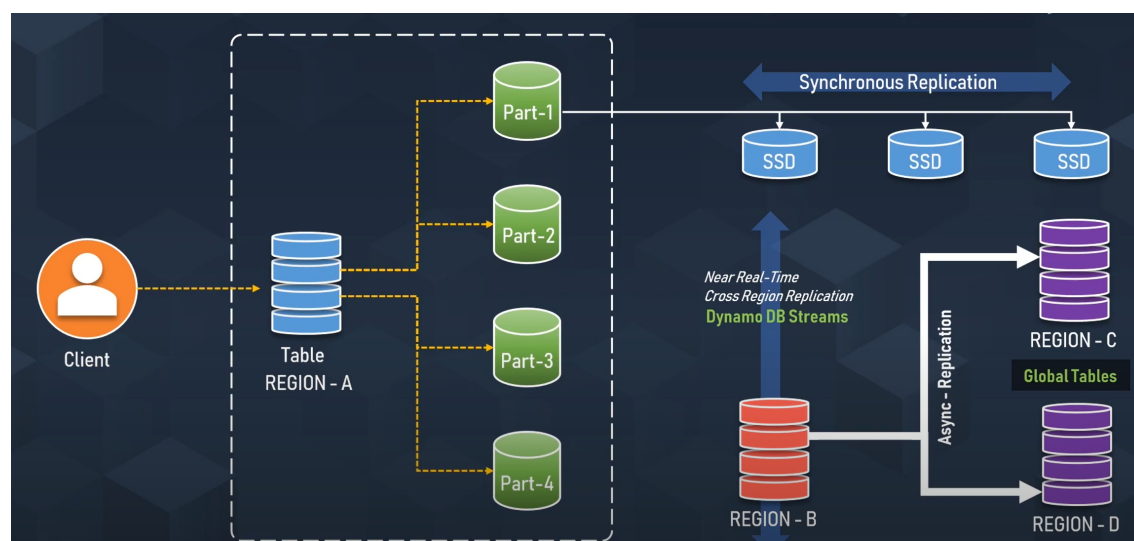
5.3 Identity Pool

I pool di identità di Amazon Cognito (identità federate) ti consentono di creare identità univoche per i tuoi utenti e federarli con i provider di identità. Con un pool di identità, puoi ottenere dei privilegi limitati e temporanei tramite credenziali AWS per accedere ad altri servizi. E' possibile creare dei ruoli con determinati livelli di permesso e renderli disponibili ad un utente autenticato.

5.4 Dynamo DB

Amazon DynamoDB è un servizio di database NoSQL interamente gestito che combina prestazioni elevate e prevedibili con una scalabilità ottimale. DynamoDB consente di ridurre l'onere di gestire e ridimensionare un database distribuito e di non doverti più preoccupare di provisioning dell'hardware, installazione e configurazione, replica, applicazione di patch al software e dimensionamento del cluster.

DynamoDB consente la crittografia dei dati inattivi, che permette di eliminare gli oneri operativi e la complessità prevista dalla protezione dei dati sensibili. Puoi creare tabelle di database in grado di archiviare e recuperare qualunque quantità di dati e soddisfare qualsiasi livello di traffico e richieste. È possibile scalare verso l'alto o verso il basso la capacità di throughput delle tabelle senza compromettere le prestazioni creando tempi di inattività. DynamoDB distribuisce automaticamente i dati e il traffico per le tabelle su un numero di server sufficiente per gestire i requisiti di throughput e storage, garantendo al contempo prestazioni rapide e costanti. Tutti i dati vengono archiviati su dischi a stato solido (SSD) e vengono replicati automaticamente in più zone di disponibilità in una regione, fornendo elevata disponibilità. Lo schema seguente mostra come avviene lo storage di una tabella in Dynamo DB.



Una tabella viene divisa in più partizioni, ogni partizione è automaticamente replicata su più dischi tenuti sincronizzati. E' anche possibile creare repliche tra regioni differinetti, con dynamodb cross region replication che vengono tenute aggiornate dalle tabelle globali.

5.5 S3

Amazon Simple Storage Service (Amazon S3) dispone di una semplice interfaccia web service che consente di archiviare e recuperare qualsiasi quantità di dati, in qualunque momento e ovunque sul Web. Amazon S3 permette di:

- Creare bucket: puoi creare e denominare un bucket in cui archiviare i dati. In Amazon S3 i bucket sono i container fondamentali per lo storage dei dati.
- Archiviare dati: puoi archiviare una quantità infinita di dati in un bucket. In un bucket Amazon S3 si può caricare una quantità illimitata di oggetti e ogni

oggetto può contenere fino a 5 TB di dati. Ogni oggetto viene archiviato e recuperato con una chiave univoca assegnata dallo sviluppatore.

- Scaricare dati: puoi eseguire il download dei dati o consentire ad altri di effettuare questa operazione. Entrambe queste operazioni possono avvenire in qualunque momento.
- Gestire Autorizzazioni: puoi concedere o negare l'accesso agli altri utenti che vogliono caricare o scaricare dati nel proprio bucket Amazon S3. Le autorizzazioni di upload e download possono essere concesse a tre tipi di utenti. I meccanismi di autenticazione possono permettere di mantenere i dati al sicuro da accesso non autorizzato.

5.5.1 Bucket

Un bucket è un container per gli oggetti archiviati in Amazon S3. Ogni oggetto è contenuto in un bucket. Ad esempio, se l'oggetto denominato `photos/puppy.jpg` è archiviato nel bucket `awsexamplebucket1` della regione Stati Uniti occidentali (Oregon), è accessibile tramite l'URL `https://awsexamplebucket1.s3.us-west-2.amazonaws.com/photos/puppy`. I bucket hanno diversi scopi:

- Organizzano lo spazio dei nomi Amazon S3 al livello più alto.
- Identificano l'account responsabile dell'archiviazione e del trasferimento dati.
- Svolgono un ruolo nel controllo degli accessi.

5.5.2 Oggetti

Gli oggetti sono le entità fondamentali archiviate in Amazon S3 e sono composti da dati e metadati. I metadati sono invece una serie di coppie nome-valore che descrivono l'oggetto. Sono disponibili alcuni metadati di default, ad esempio la data dell'ultima modifica, e metadati HTTP standard, come Content-Type. È anche possibile specificare metadati personalizzati al momento dell'archiviazione dell'oggetto. Un oggetto viene identificato in modo univoco in un bucket tramite una chiave (nome) e un ID di versione.

5.5.3 Chiavi

Una chiave è un identificativo univoco di un oggetto in un bucket. Per ogni oggetto in un bucket è presente una e soltanto una chiave. Una chiave e un ID versione identifica in modo univoco ciascun oggetto. Quindi puoi pensare ad Amazon S3 come a una mappa di dati di base tra "bucket + chiave + versione" e l'oggetto stesso. Si può fare riferimento in modo univoco a ogni oggetto in Amazon S3 tramite la

combinazione di endpoint del servizio Web, nome del bucket, chiave e, facoltativamente, versione. Ad esempio, nell'URL <https://doc.s3.amazonaws.com/2006-03-01/AmazonS3.wsdl>, "doc" è il nome del bucket e "2006-03-01/AmazonS3.wsdl" è la chiave.

6 Sviluppo di un'applicazione Serverless: AWSServerlessApplication

Nella seguente sezione viene presentata `AWSServerlessApplication`, è un applicazione di esempio modulare affine alla gestione degli utenti, integrabile in un qualsiasi scenario che preveda l'utilizzo di un account. L'applicazione mette a disposizione delle API a supporto delle fasi di registrazione, login, cambio password, recupero della password e permette di eseguire le operazione CRUD(create, read, update, delete) legate all'utente. L'applicazione è un semplice backend che ci permette di capire i meccanismi di creazione e distribuzione di un progetto che segue il paradigma Serverless e utilizza le tecnologie AWS sopra descritte. L'applicazione è sviluppata in .NET core 3.1 e pubblicata sul cloud AWS, fruibile dall'url <https://7ge1zdatk1.execute-api.eu-west-1.amazonaws.com/Prod/swagger/index.html>. Assieme a questa applicazione è stata prodotta "ds-baas-application" al fine di fornire, anche, solo un banale esempio di un' applicazione Baas e per svolgere alcuni semplici test di correttezza del funzionamento dell'applicazione `AWSServerlessApplication`. In seguito verranno descritti gli aspetti maggiormente significativi legati allo sviluppo dell'applicazione.

7 Requisiti

Nelle sezioni successive vengono presentati i requisiti funzionali e non funzionali legati allo sviluppo di queste applicazioni.

7.1 Requisiti funzionali

Per la configurazione e il deploy, anche detto pubblicazione, di questi progetti è necessario essere in possesso di credenziali AWS e di un ruolo con i permessi di accesso ai vari servizi. Viene ritenuto fondamentale per l'integrazione e il funzionamento dei vari servizi un approccio event-driven. Il sistema deve garantire un meccanismo di registrazione, con scelta di una password personale post registrazione; deve garantire la possibilità di cambio della password e il suo ripristino qual'ora l'utente se la sia dimenticata. Il sistema deve gestire le informazioni relative all'utente e permetterne la modifica.

7.2 Requisiti non funzionali

Gli esempi presentati nel progetto, mirano a garantire scalabilità, disponibilità, e facilità d'uso per lo sviluppatore. Il sistema deve garantire consistenza all'interno dell'architettura e permettere un approccio a micro-servizi, per farlo deve essere modulare, completamente riusabile e customizzabile a seconda del contesto.

7.3 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.

7.4 Self-assessment policy

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

8 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 / technology is the best suited to face this project's requirements? What's the abstraction gap among the available models / paradigms / technologies and the problem to be solved?

9 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 evaluated during the design phase is welcome too.

9.1 Structure

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

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

9.2 Behaviour

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

9.3 Interaction

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

10 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. In this case, the produced documentation should be referenced here.

11 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.

12 Deployment Instructions

Explain here how to install and launch the produced software artefacts. Assume the software 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.

13 Usage Examples

Show how to use the produced software artefacts.

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

14 Conclusions

Recap what you did

14.1 Future Works

Recap what you did *not*

14.2 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} , \mathbf{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.