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.

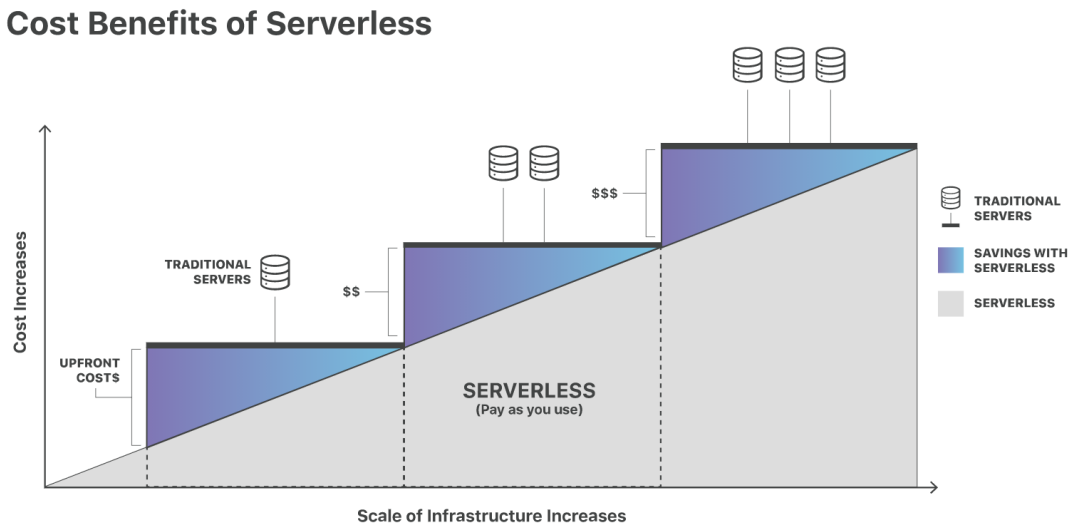
2 Architetture e paradigma Serverless

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, 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, cosa non solo è possibile ma, in determinate circostanze, è desiderabile per un'organizzazione eseguire un servizio FaaS su server che non 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, 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: "ci si accorgerà che è una bugia".[1] 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 del mantenimento o della configurazione. La spiegazione del successo di questa tecnologia è mostrata nel seguente grafico.



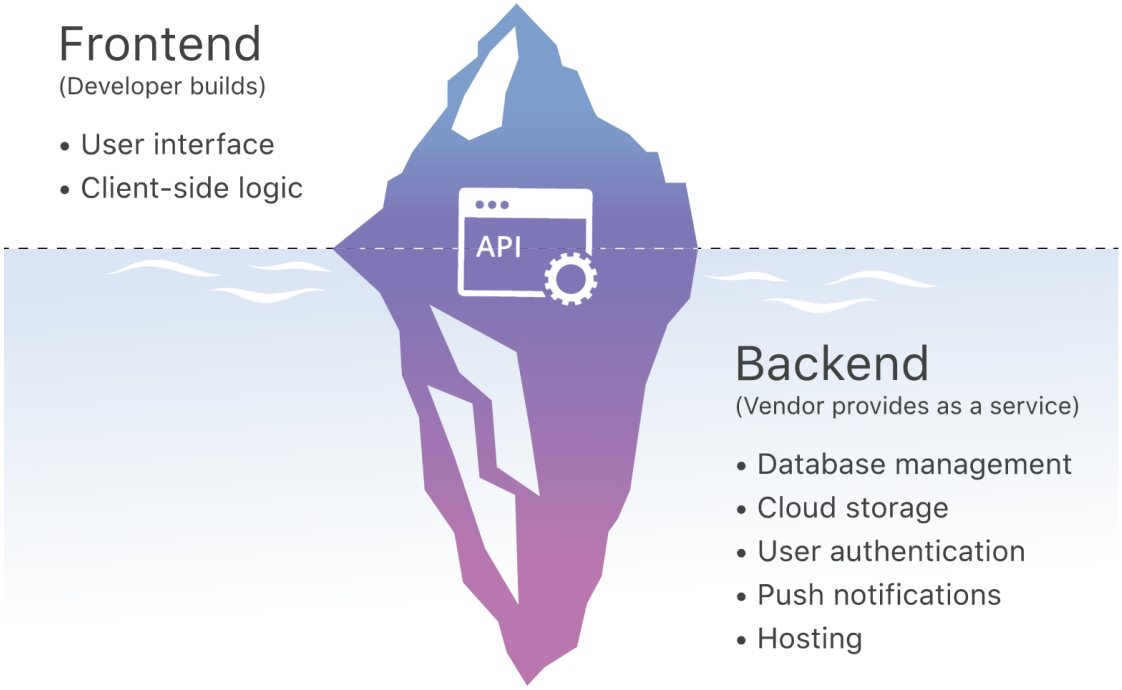
Dal grafico è evidente che il costo iniziale è maggiore, ma se paragonato al costo dell'allestimento di un infrastruttura server fisica interna o in cloud è inferiore e questo trend rimane anche al crescere della dimensione dell'infrastruttura. La differenza sta nel pagare le risorse in base all'elaborazione.

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.

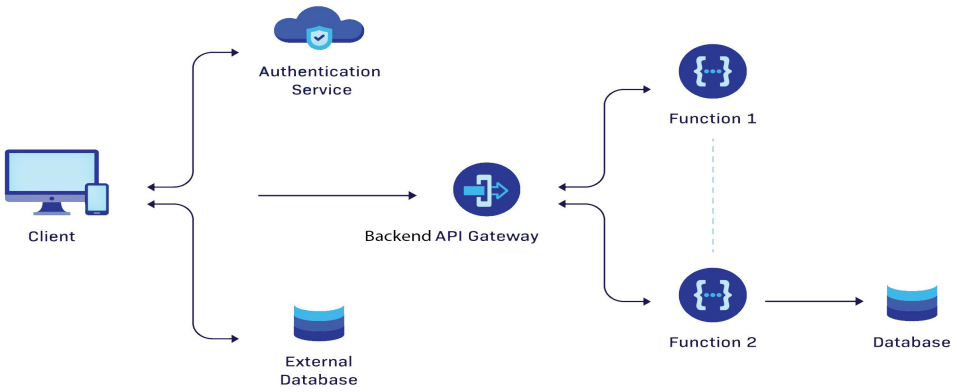
2.3 Backend as a Service

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 le proprie API che devono essere incorporate singolarmente in una app, un processo che può richiedere molto tempo per gli sviluppatori. 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 a cui 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 poiché riguarda in particolare le esigenze del cloud computing degli 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 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 containers stateless; le funzioni gestiscono la logica dell'applicazione server, gli stati vengono gestiti utilizzando dei servizi esterni. Questo modello consente agli sviluppatori di creare, eseguire e gestire i pacchetti applicativi come funzioni, senza doversi occupare della loro infrastruttura. FaaS è anche una modalità di esecuzione dell'elaborazione Serverless, con la quale gli sviluppatori scrivono la logica delle loro applicazioni che viene eseguita in containers totalmente gestiti

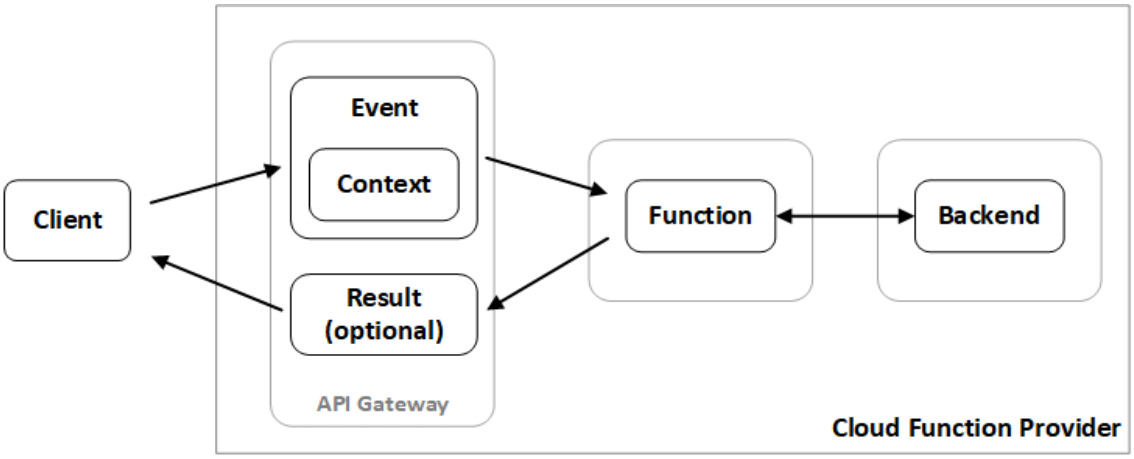
dalla piattaforma. In genere questa piattaforma si trova nel cloud, ma il modello si sta espandendo per integrare deployment on premise e ibridi. Nel modello Serverless, gli sviluppatori non si occupano dei problemi legati all'infrastruttura, come la gestione, il provisioning dei server o l'allocazione delle risorse. 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 e restituire il risultato. Non sempre è presente un risultato da inviare al client come risposta.

2.5 Vantaggi e svantaggi

Abbiamo già delineato i vantaggi che generano le architetture FaaS e quelle BaaS, combinandoli possiamo otterenere i vantaggi che offre l'approccio Serverless, riassunti 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 metto a disposizione 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 un team di sviluppatori lavora con i mainframe. Inizialmente c'erano un insieme di tecnologie raccolte sotto il nome di TPM (Transaction Process Monitoring), ad esempio, CICS (Customer Information Control System) di IBM e tutto ciò che faceva era gestire elementi non funzionali. TPM si occupa della gestione problemi relativi al carico, alla sicurezza, al monitoraggio, alla distribuzione, ecc. Quindi, TPM può essere definito come un antenato del Serverless.

3.1 Confronto con Infrastructure as a Service(IaaS)

A differenza del Serverless, IaaS offre un'infrastruttura completa, come richiesto dagli sviluppatori, che devono prima installare una macchina virtuale, configurarla per poi distribuire l'app. Con IaaS gran parte della gestione dell'infrastruttura ricade sugli sviluppatori.

3.2 Confronto con Container as a Service(CaaS)

A differenza del Serverless, 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 ciò permette di 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)

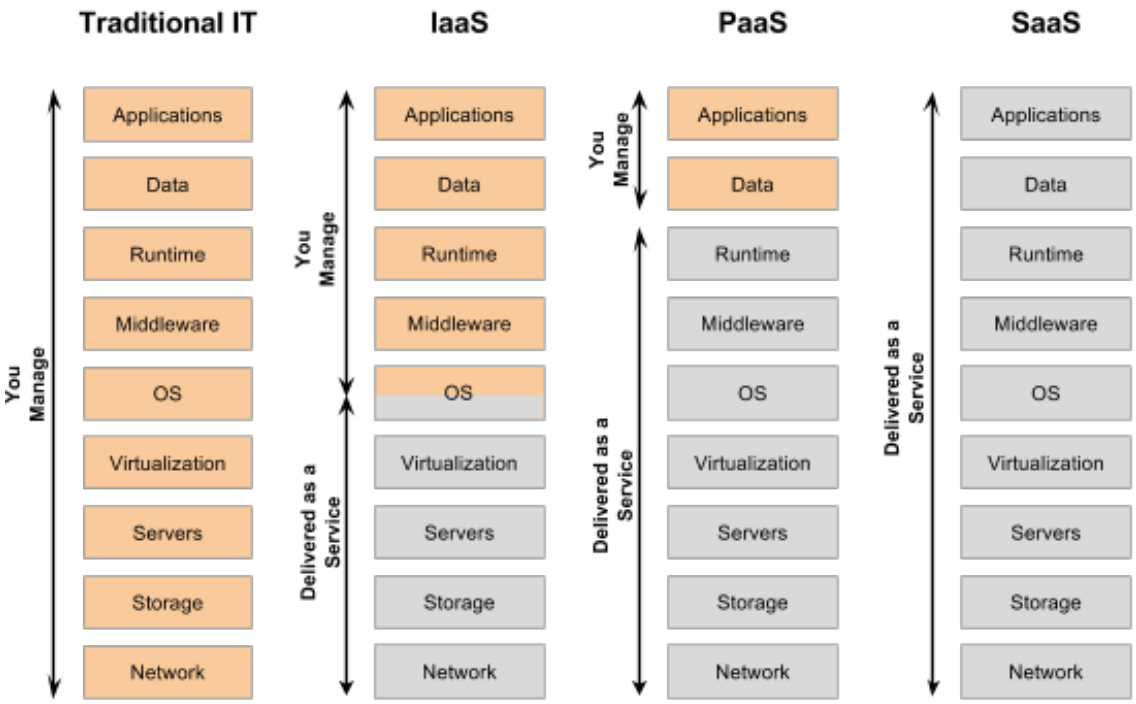
Poi è arrivato il momento di concentrarsi su PaaS, che è stato un passo verso il Serverless, è la versione migliorata di CaaS, in cui lo sviluppatore non deve preoccuparsi del sistema operativo e del suo funzionamento. Lo sviluppare si limita a distribuire l'app o il servizio sulla piattaforma. La differenza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

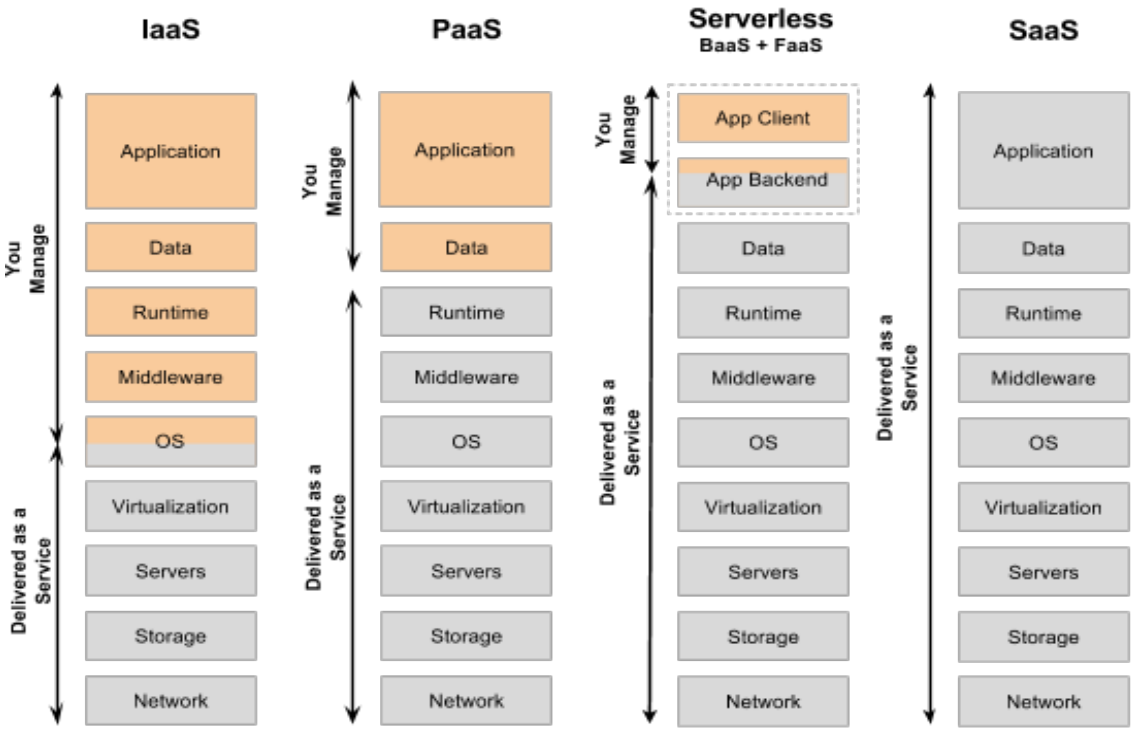
I container lavorano su un'unica macchina, alla quale sono assegnati inizialmente; possono, però, essere riposizionati. L'assegnazione del server in Serverless è responsabilità del provider, e viene attaccata quando un evento attiva la funzione. In Serverless, non è confermato che lo stesso scope verrà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



E' evidente che l'architettura tradizionale ad oggi è 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

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 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 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 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 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 microservizi. 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 microservizi, 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 microservizi 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), si crea una copia di backup della build precedente e si va a pubblicare la nostra applicazione mandando 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 ndl 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

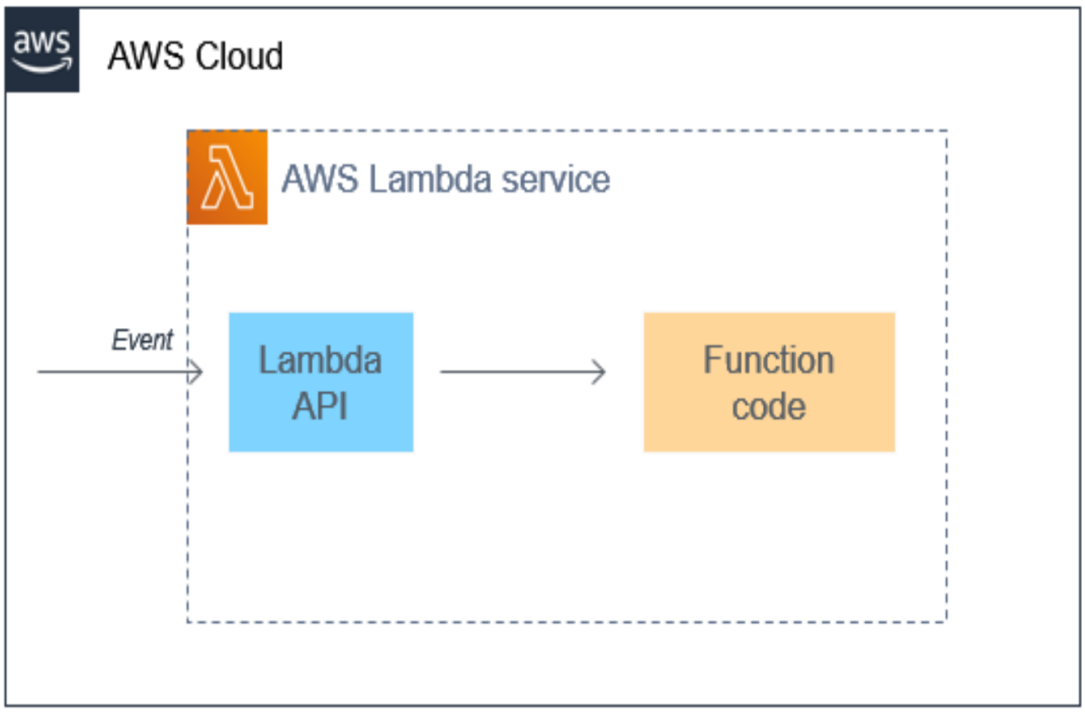
negli interessi dello stesso Amazon farci conoscere come fanno i suoi servizi a funzionare, pertanto si fa veramente fatica a trovare qualcosa di certo che ci permetta di capire le architetture interne e fare reverse engineering sui servizi. Per erogare servizi basati su container si ipotizza vengano utilizzati Docker e Kubernetes o tecnologie che permettano servizi simili. I servizi presentati in seguito sono tutti servizi che AWS mette a disposizione dei propri utenti, ogni servizio è a sè stante ed indipendente dagli altri; questo insieme di servizi ci permette di creare delle Serverless application. Lambda è il servizio fondamentale che costituisce la parte FaaS della nostra applicazione e Cognito, Dynamo DB e S3 sono dei servizi che ci permettono di integrare la parte BaaS al fine di costruire una Serverless application. Concentriamoci su una panoramica generale degli aspetti principali che ogni servizio offre in modo da capire come utilizzarli ed intergrarli 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 sull' 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, come 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 microservizi 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 microservizio, 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.[2]

5.1.1 Lambda function

Una lambda function è quanto abbiamo descritto fin ora. E' importante sapere che AWS Lambda offre supporto nativo a codici Java, Go, PowerShell, Node.js, C#, Python e Ruby e fornisce un'API Runtime che consente di utilizzare qualsiasi altro linguaggio di programmazione per creare le tue funzioni. Queste Lambda eseguono delle funzioni della nostra applicazione in seguito a delle specifiche chiamate Invoke, in risposta a degli endpoit delle API a seguito della configurazione di API gateway o in seguito a degli eventi specifici interni ai sistemi AWS come mostrato per

Cognito nella sezione Dettagli implementativi dove viene mostrato come il servizio AWS Cognito mette a disposizione un insieme di eventi a cui è possibile associare una Lambda e quindi far si che quegli eventi siano gli entrypoint di una Lambda. Per ulteriori dettagli si rimanda al capito Dettagli Implementativi.

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 codice in più funzioni viene fatta automaticamente e in maniera ottimizzata da AWS. E' possibile riprodurre una Lambda application, ovviamente con prestazioni peggiori, creando n Lambda functions è configurando a mano l'API Gateway per invocarle a seconda della route scelta. Inoltre vengono definite Lambda application quelle funzioni Lambda che integrano altri servizi.

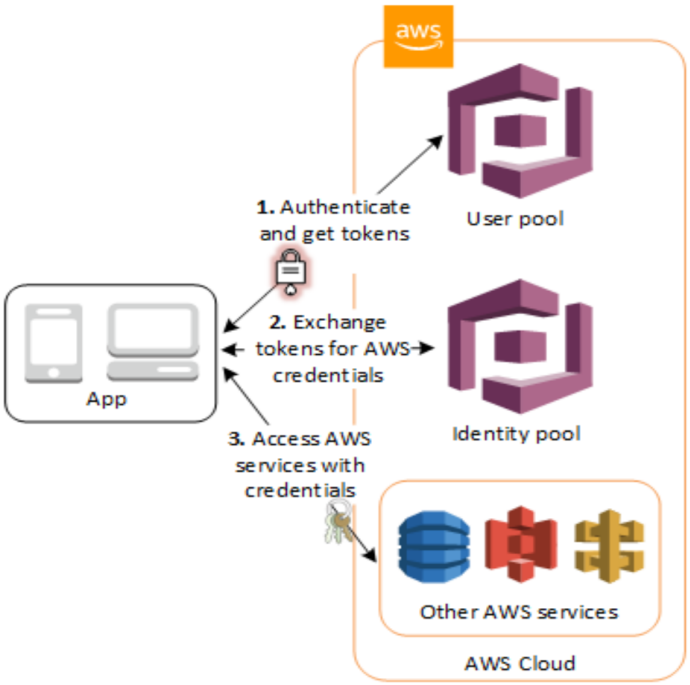
5.2 Cognito

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. Il 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 Dynamo DB.[3]

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:

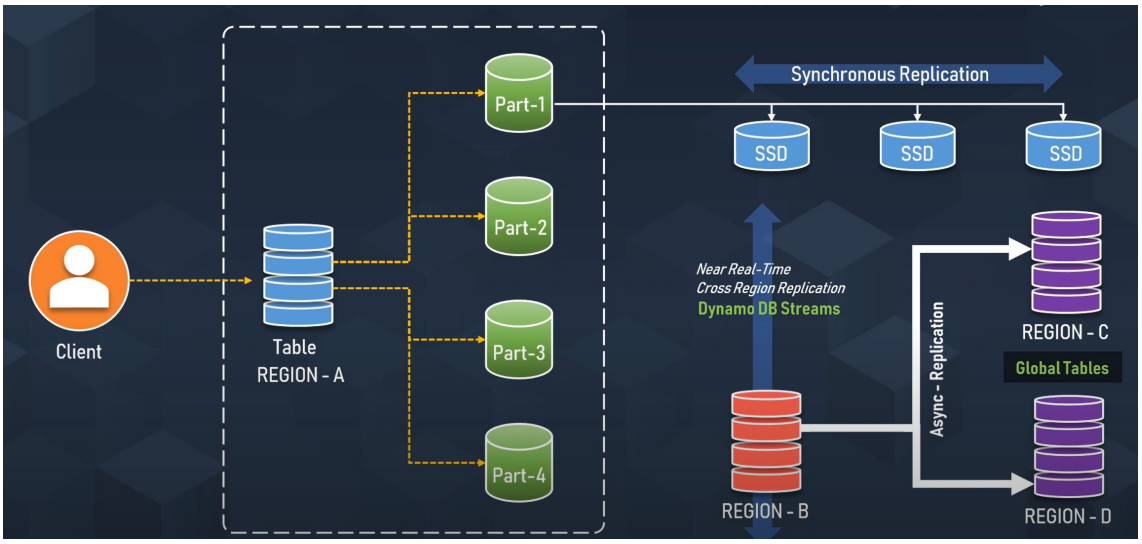
- Servizi di registrazione e di accesso.
- Un'interfaccia utente Web integrata personalizzabile per l'accesso degli utenti.
- Accesso social tramite Facebook, Google, Login with Amazon e Accesso con Apple e accesso tramite i provider di identità SAML dal pool di utenti.
- Gestione della directory utente 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 JSON Web token (JWT) che è possibile utilizzare per proteggere e concedere le autorizzazioni di accesso alle proprie API o per lo scambio di credenziali AWS.[3]

5.3 Identiy 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.[3]

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.[4] 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 differineti, 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.[5] 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 S3. I meccanismi di autenticazione possono permettere di mantenere i dati al sicuro dagli accessi non autorizzati.

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.jpg.[5] 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]

5.5.3 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.[5]

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 utente. 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 semblice backend che ci permette di capire i meccanismi di creazione e distribuizione di un progetto che segue il paradma Serverless e utilizza le tecnologie AWS sopradescritte. 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 permettergli di modificarle.

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 microservizi, per farlo deve essere modulare, completamente riusabile e customizzabile a seconda del contesto.

7.3 Scenari

Il progetto è solo esemplificativo e si pone come obbittivo quello di aiutare a comprendere come funzionano le tecnologie sopradescritte, quali sono le loro potenzialità e la facilità di utilizzo. L'esempio è comunque modulare e fornisce una completa gestione degli utenti, è integrabile ed estendibile a qualunque applciazione web o mobile che necessiti queste funzionalità. Si raccomanda però di integrare questa applicazione in progetti che utilizzano lo stesso paradigma, aggiungendo nei file di configurazione, l'endpoint del nuovo modulo pubblicato o viceversa inserndo l'endpoint di questo modulo a quello nuovo, così risulta banale far comunicare le due applicazioni. Seguendo questo approccio si rispetta la modularità del sistema e si crea un' architettura a microservizi. Attualmente l'utente può utilizzare il sistema usufruendo delle API REST a disposizione. Le chiamate consentono di:

- registrare un utente;
- cambiare la password;
- effettuare il login;
- ripristinare la password;
- aggiornare i suoi dati;
- eliminare l'utente;
- riceve il dettaglio di un profilo utente;
- ricevere il dettaglio di tutti i profili utente registrati;

7.4 Criteri di valutazione

La qualità del software prodotto dovrebbe basarsi sui principi di modularità, riusabilità e integrabilità; nonchè, sulla qualità del codice stesso.

L'efficenza del progetto dovrebbe basarsi su disponibilità, coerenza, tolleranza di partizione e confidenzialità. Va dato rilievo anche alle semplificazioni che l'applicazione di questo paradigma garantisce agli svliluppatori. Ai fini dei processi produttivi è molto rilevante valutare anche il lato economico.

8 Analisi dei Requisiti

Il paradigma Serverless può essere adottato anche utilizzando altre tecnologie sopracitate. AWS permette di applicare a pieno il paradigma Serverless e mette a disposizione tantissimi strumenti e servizi che favoriscono questo approccio, attualmente è quello che rende più netto il livello di astrazione che il paradigma crea. Per poter utilizzare al meglio queste tecnologie è necessario conoscerle e fare uso della documentazione per poter configurare i servizi. E' fondamentale imparare a definire i servizi in gioco e il loro funzionamento a partire dal Serverless template laddove possibile.

9 Design

Il progetto è molto semplice il design usato non rispetta alcun pattern di stile formale essendo un progetto di esempio; vi è una divisione logica degli elementi presenti nel progetto.

9.1 Struttura

La struttura del progetto è generata a partire dal modello di progetto Serverless fornito con l'installazione del SDK. I servizi in gioco ed eventuali Lambda functions aggiuntive vengono definite nel file Serverless.template; ricordiamo che a seguito della pubblicazione sarà l'API gateway, in automatico a suddividere il nostro progetto in Lambda functions. Il progetto presenta poi dei file di configurazione dove vengono definite le informazioni relative ad alcuni servizi. Il Controller definisce le API REST dell'applicazione. I Models definisco lo User, le classi di richiesta provenienti dal frontend e le classi che modellano i dati presenti sulle tabelle di DynamoDB. I services raggruppano interfacce e classi che modellano le logiche dietro ai controller. Le extensions sono funzionalità di supporto per la gestione dei dati. Le utils contengono alcune classi con funzionalità di supporto alle logiche generali. Gli helpers sono classi di supporto alle configurazioni dei services nello startup. I profiles definisco i mapping tra le classi. I Dto Data Transfer Object vengono usati come tipi di ritorno dai controller. Authentication fornisce le classi

per l'autenticazione con Cognito e la gestione dei token. AWS contiene l'injector per passare le informazioni che permettono l'uso dei servizi nelle Lambda e le Lambda aggiuntive. HTML contiene i modelli presenti su S3 che rappresentano i template delle mail che vengono inviate tramite Lambda.

9.2 Comportamento

Il comportamento è determinato dalla chiamata, il controller chiama un service che si occupa di svolgere la logica relativa alla chiamata, inclusa l'interazione con Dynamo DB. Gli eventi d'invio e ripristino della password sono gestiti da Cognito che va ad invocare la nostra CognitoCustomMessageFunction. Il service che gestisce la chiamata api/Users va a invocare un evento che scatena la getUsersLambdaFunction per ottenere la lista di tutti gli utenti.

9.3 Interazione

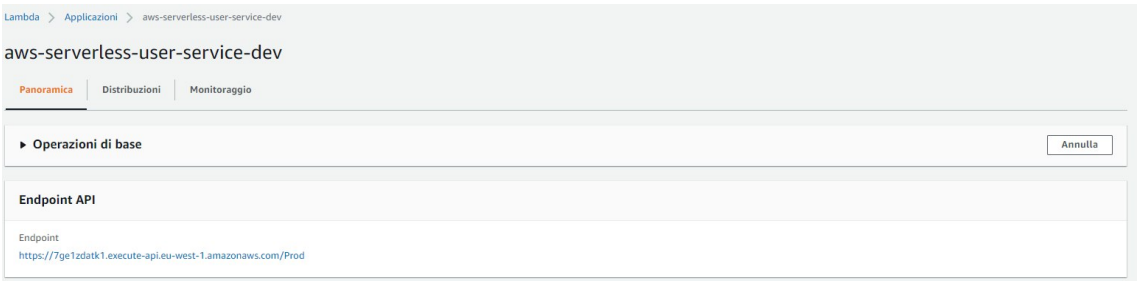
Le interazioni sono dettate dall'utente che può servirsi dello swagger, o di postman per effettuare le chiamate. Si ribadisce che è l'APIgateway a mascherare le interazioni tra le varie funzioni che esso stesso crea per suddividere l'applicazione.

10 Dettagli implementativi

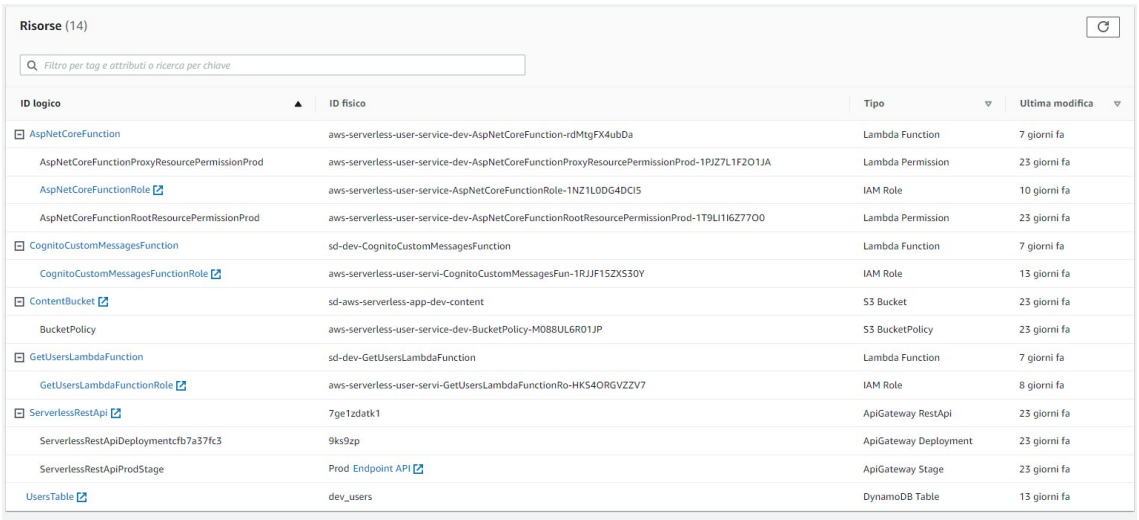
In questo capito vengono presentate le API REST che AWSServerlessApplication mette a disposizione e viene fatta luce sulle Lambda function e dei servizi che ne permettono il funzionamento. Il seguente elenco mostra l'insieme delle API REST che l'applicazione mette a disposizione:

- POST /api/Users/signIn
- PUT /api/Users/setPassword
- GET /api/Users/id
- PUT /api/Users/id
- DELETE /api/Users/id
- GET /api/Users
- POST /api/Users
- GET /api/Users/forgotPassword/email
- PUT /api/Users/resetPassword

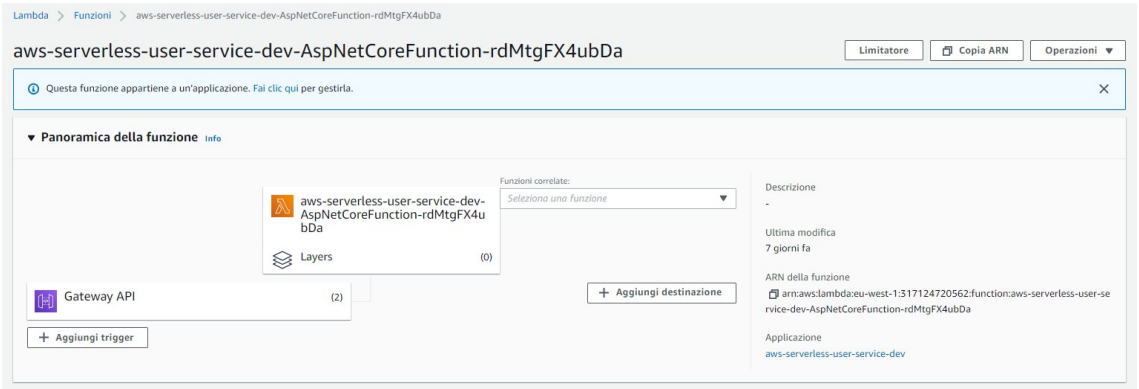
La figura seguente mostra la Lambda application costruita dal template dell'applicazione AWSServerless.



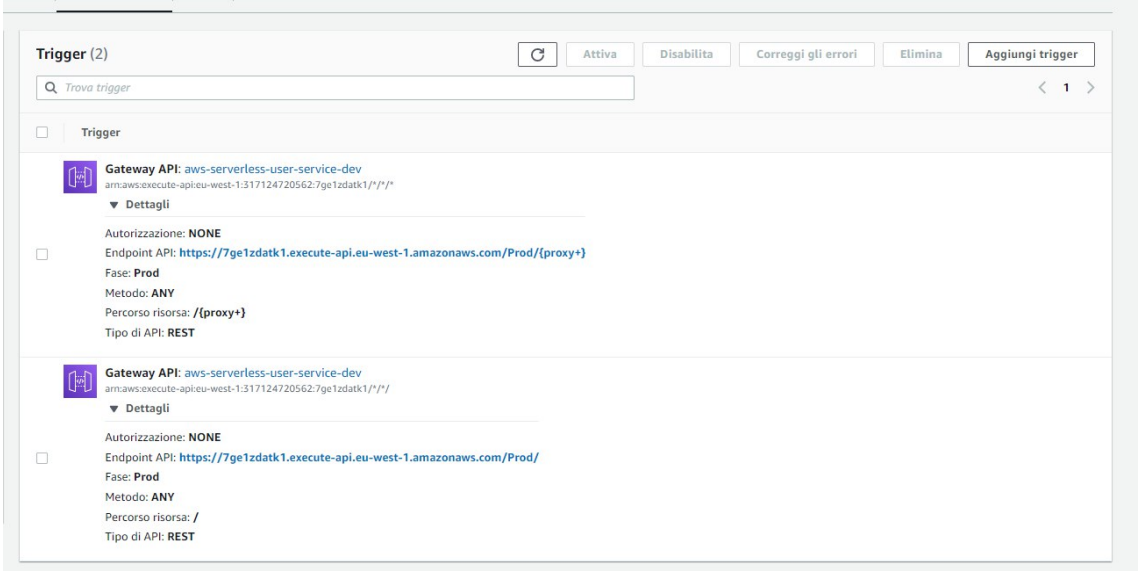
L'immagine seguente mostra invece le risorse correlate ad AWSServerless application definite nel template.



E' interessante approfondire la AspNetCoreFunction, che viene mostrata nella figura seguente.



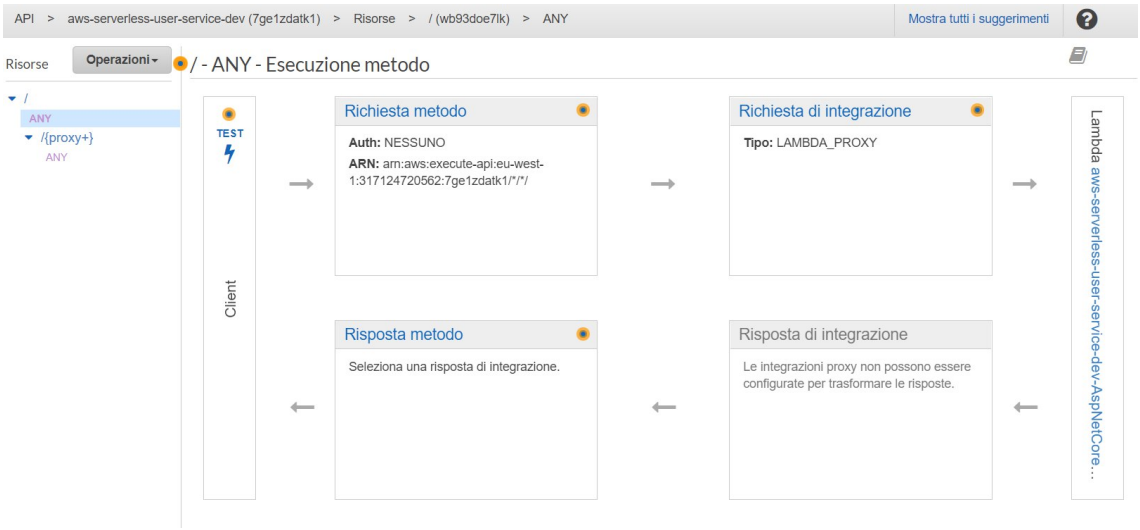
Come evidente dall'immagine è l'API gateway a fungere da trigger a questa funzione. Ed è possibile constatarlo anche nella voce trigger, come mostrato nell'immagine seguente.



Questa funzione Lambda maschera l'esecuzione dell'intera Serverless application. Nella definizione infatti questa funzione prevede come trigger:

```
"Events": {
        "ProxyResource": {
          "Type": "Api",
          "Properties": {
            "Path": "/{proxy+}",
            "Method": "ANY"
          }
        },
```

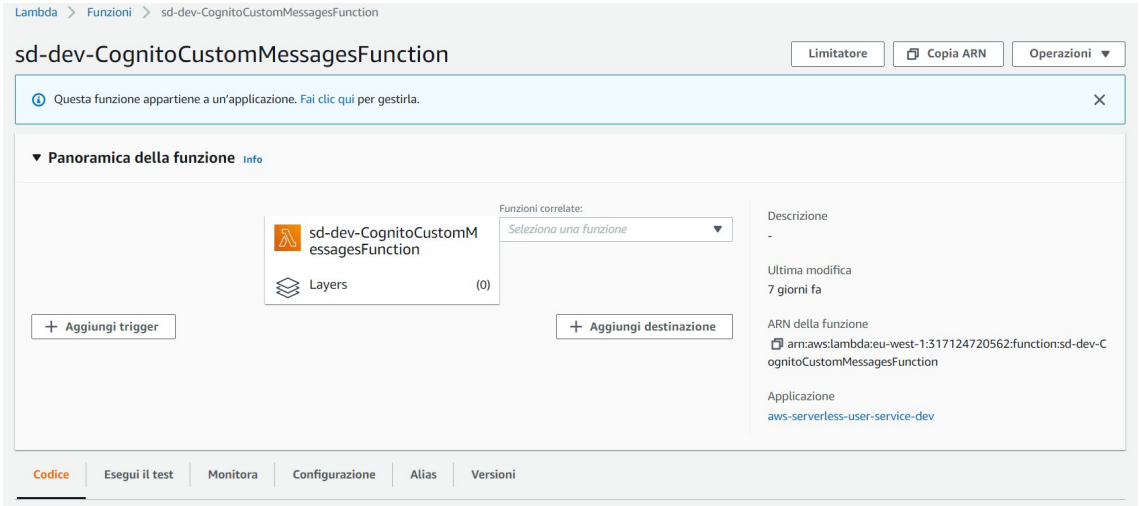
Possiamo riscontrarlo anche dal cloudFormation, come mostrato nell'immagine seguente.



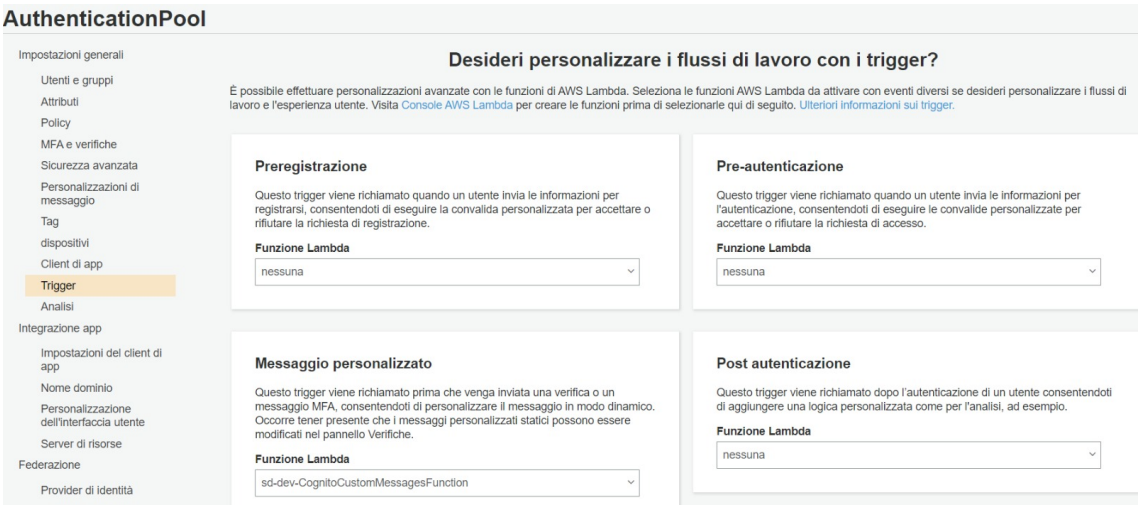
AspNetCoreFunction maschera ciò che viene fatto internamente, è l'API gateway a suddividere il comportamento della nostra applicazione in questa Lambda. Ogni volta viene scatenata solo la parte di codice necessaria per l'esecuzione dell'evento che in questo caso corrisponde alla chiamata API su una delle route sopradescritte. Le chiamate API sopracitate si integrano con Cognito per la fase di autenticazione e con Dynamo DB per la gestione dei dati relativi agli utenti.

10.1 Cognito custom message function

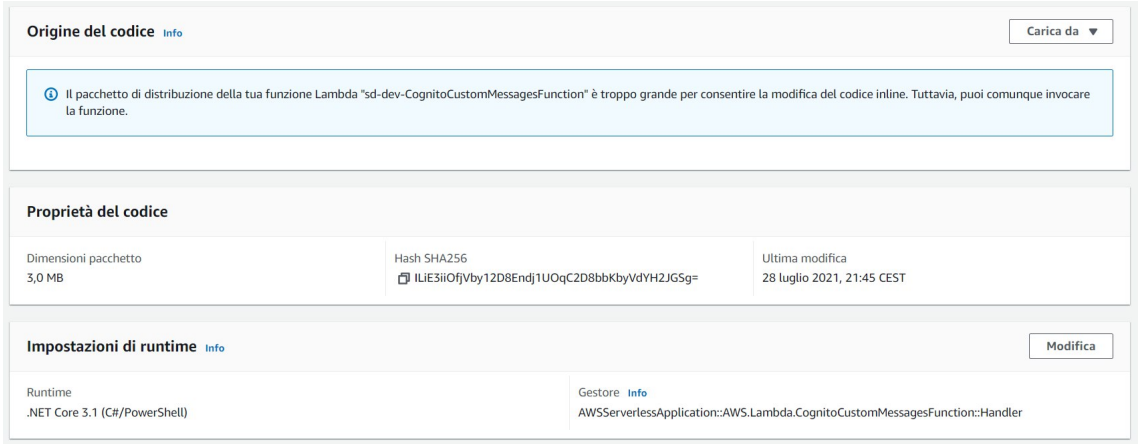
Questa è una Lambda definita nel template che permette di personalizzare i messaggi inviati da Cognito agli utenti in varie fasi del processo di registrazione e autenticazione. I template delle mail che vengono inviati sono memorizzati in un Bucket S3. La lambda è costruita come mostrato nella figura seguente.



Ed esegue il codice presente nella classe CognitoCustomMessagesFunction.cs. Essendo che gran parte dei servizi AWS segue il paradigma event-driven è molto facile costruire applicazioni Serverless, ad esempio è interessante notare come Cognito metta a disposizione una sezione Trigger dove vengono definiti una serie di eventi ai quali è possibile associare una Lambda, come mostrato nella figura seguente.

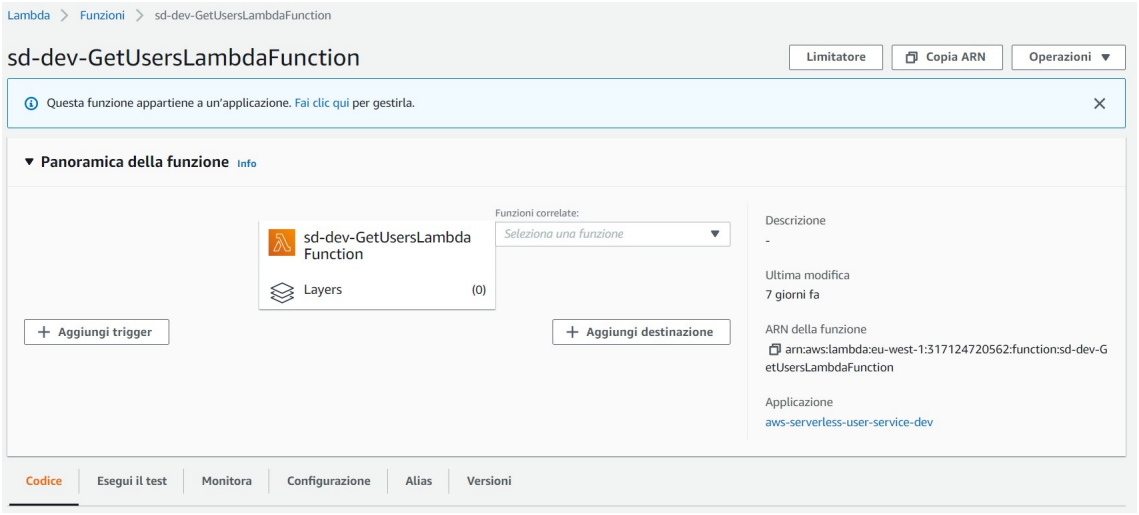


Questo ci permette anche di non associare il trigger alla Lambda sulla Lambda stessa. E' importante però notare che ogni Lambda ha un hendler che ne permette l'attivazione. Ciò è visibile dal template, dalla classe che ne definisce il comportamento o anche dalla console AWS selezionando la Lambda come mostrato nella seguente figura.



10.2 Get users functions

Questa Lambda è definita nel template ed è un esempio di come è possibile scorporare l'esecuzione di parte del nostro codice dal flusso principale. Il comportamento di questa Lambda è definito in GetUsersLambdaFunction.cs ovvero è una Scan di Dynamo DB che ci permette di visualizzare tutti gli utenti presenti a database. L'immagine seguente ne mostra la struttura.



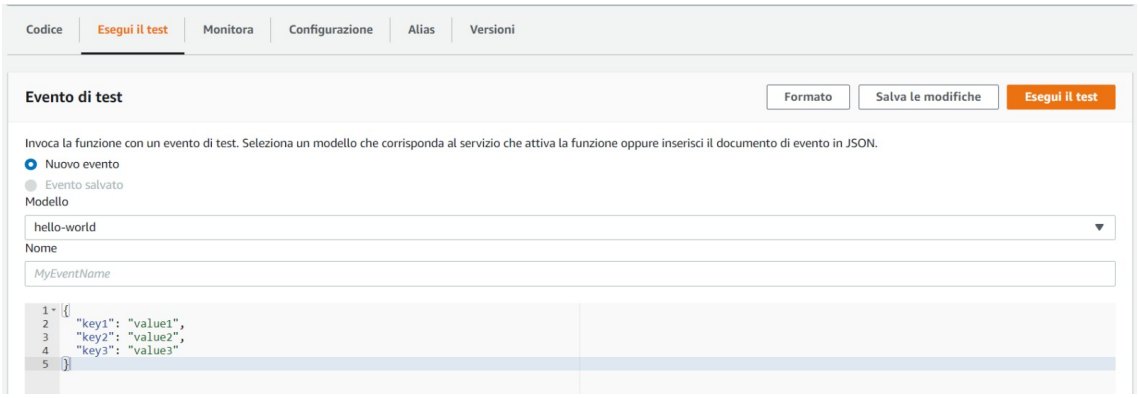
Questa chiamata è asincona ed essendo già in esecuzione su una Lambda è possibile invocarla come segue:

```
var response = await _amazonLambda.InvokeAsync(new InvokeRequest
    {
      FunctionName = "sd-dev-GetUsersLambdaFunction",
     });
var users = await
JsonSerializer.DeserializeAsync<List<User>>(response.Payload);
```

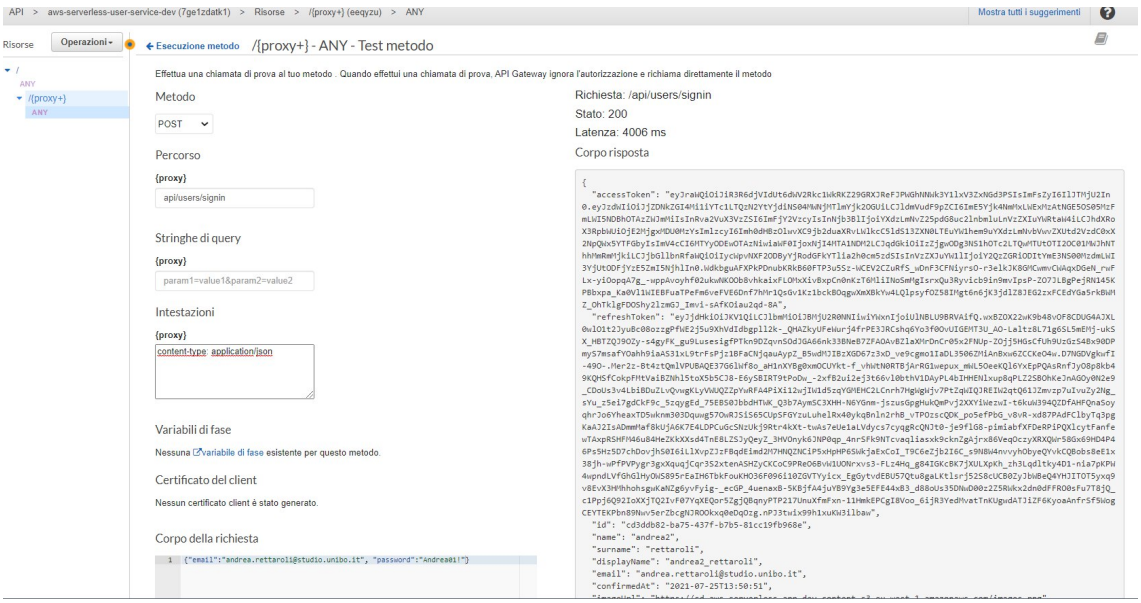
Questa esecuzione è solo esemplificativa e non rispetta le best practice di Lamdba e del paradigma Serverless, infatti il suo flusso di elaborazione aumenta i tempi di esecuzione. E' importante però sottolineare come sia possibile lanciare esecuzioni sincrone al flusso principale che non debbano ritornare un risultato, ad esempio la PUT di un oggetto a database.

11 Autovalutazione e Convalida

I test sono stati eseguiti per le funzionalità Lambda con l'apposito servizio messo a disposizione sulla console AWS, come mostrato nella figura seguente.



Mettendo nella Lambda anche solo un Log è possibile verificare che realmente la sua pubblicazione sia andata a buon fine e che la Lambda pubblicata stia funzionando correttamente. Per testare il funzionamento generale dell'applicazione è stato principalmente utilizzato lo swagger. Risulta inoltre possibile testare le API direttamente dall'API gateway come mostrato nella figura seguente.



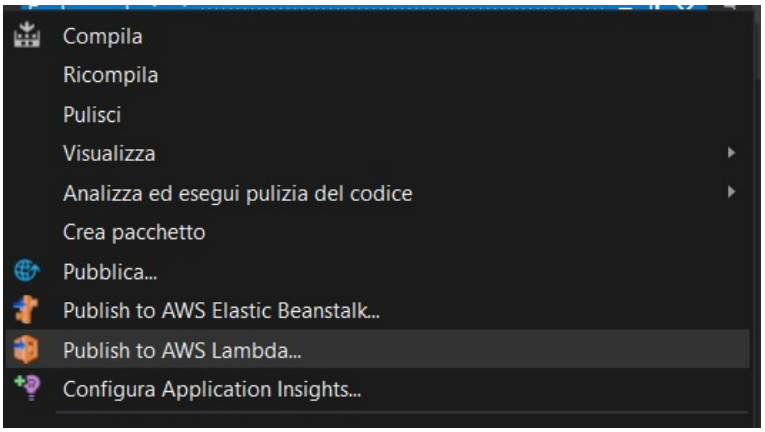
Come test di correttezza del pool di utenti e della configurazione di Cognito è stata sviluppata l'applicazione ds-baas-application in React JS con l'utilizzo di AWS-Amplify. L'applicazione si occupa di effettuare il login dell'utente direttamente su Cognito e poi una volta ottenuto il token effettua la chiamata get dell'utente come prova per verificare la correttezza della login.

12 Istruzioni per il Deploy

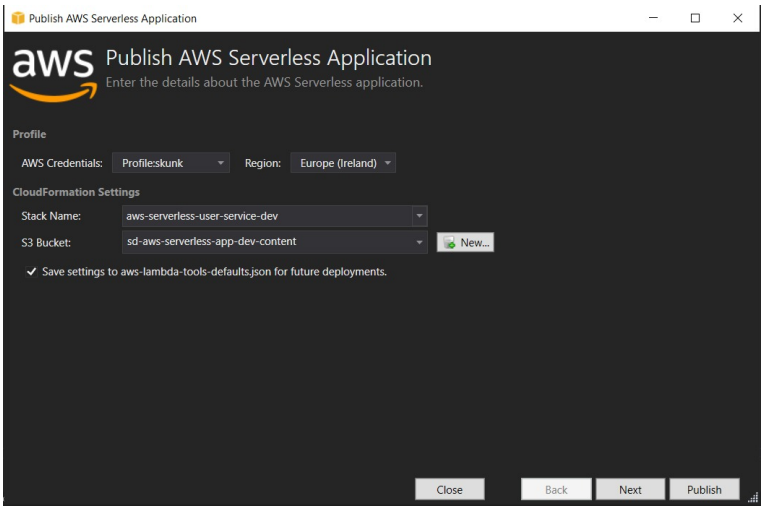
Per conseguire il deploy correttamante è necessario:

- essere in possesso di credenziali AWS con un ruolo che abbia sufficienti permessi per poter pubblicare correttamente tutti gli elementi definiti nel template.
- aver configurato correttamente il Serverless.template .

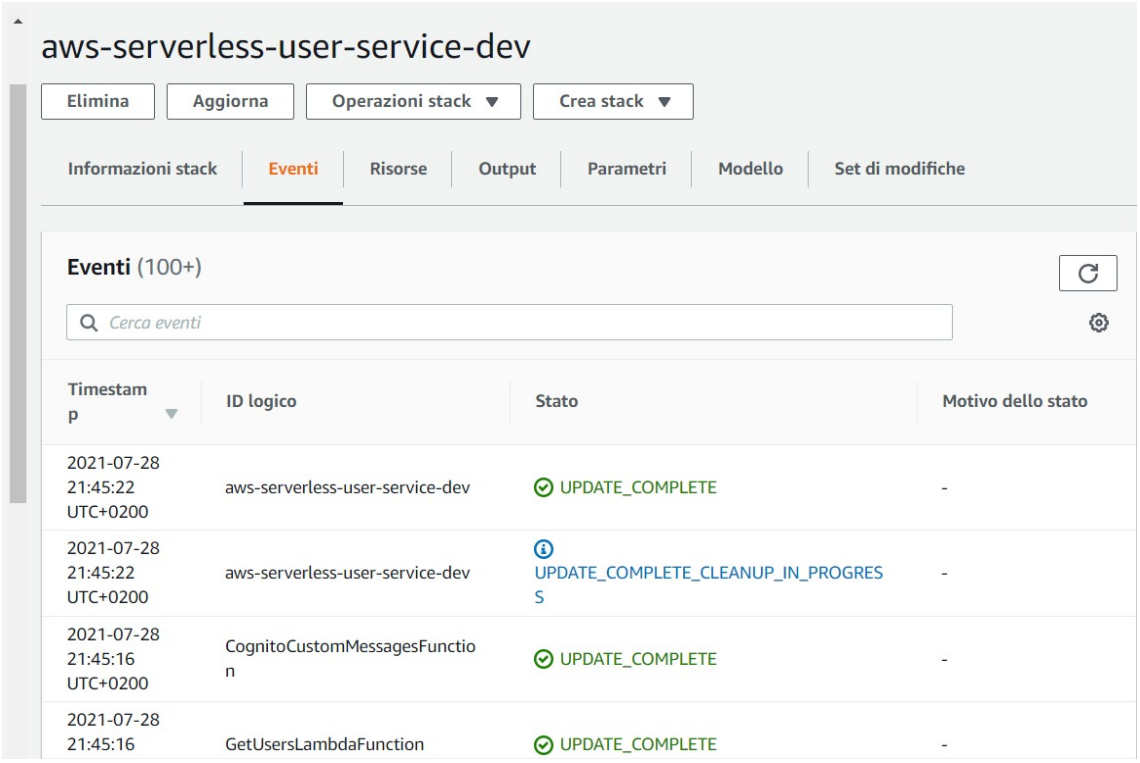
Senza il primo requisito non è possibile conseguire il deploy. Una volta in possesso di credenziali e con i profili ben configurati (nel file credentials presente nella cartella .aws dentro il proprio profilo utente) è possibile fare tasto destro sul progetto e selezionare la voce "publish to AWS Lambda..", come mostrato nella figura seguente.



Ora dovrai selezionare il profilo definito dalle credenziali, la regione e un bucket S3, come mostrato nella figura seguente; facendo next potrai anche definire l'env o altri fattori.



Premi il tasto pubblica per pubblicare, se stai utilizzando visual studio poi vedere l'aggiornamento in tempo reale e così l'esito della tua pubblicazione; la best practice è andare sulla console AWS, cercare il servizio CloudFormation e controllare da li se tutto è andato a buon fine, come mostrato nella figura seguente.



Si ricorda che in caso di fallimento viene automaticamente fatto un rollback da parte di AWS alla pubblicazione precedente e il servizio è sempre e comunque disponibile.

13 Esempi di utilizzo

L'applicazione è modulare e come già evidenziato può essere utilizzata in qualunque applicazione che prevede degli utenti da gestire come ad esempio: e-commerce, portali aziendali, social network, ecc.. .

14 Conclusioni

In seguito all'approfondimento, si sostiene che il paradigma Serverless abbia rivoluzionato le tradizionali metodologie di sviluppo, permettendo di creare applicazioni server distribuite; fornendo un livello di astrazione in più allo sviluppatore che lo libera di tutta la parte di devOps del sistema architetturale che sostiene l'applicazione. Si pensa anche che questo nuovo paradigma di sviluppo proietti il software verso il futuro dando un valido supporto alla progettazione a microservizi. Il rapporto costi benefici è ampiamente positivo e lo attesta la crescita esponenziale che questo nuovo modo di sviluppare sta avendo. Si ritiene che il vincolo del cloud provider,

a volte, possa essere limitante; però si pensa che in futuro verranno fatti maggiori passi verso l'apertura e l'integrazione tra vari cloud provider; inoltre se si effettua la giusta scelta del provider a seconda delle esigenze questo vincolo diventa completamente tollerabile. A seguito dell'analisi e dello studio di questo padigma, e dell'applicazione dei concetti appresi, si afferma che:

- vi è un forte riscontro tra la teoria e la pratica;
- se si conoscono i servizi messi a disposizione dal cloud provider scelto è molto semplice rispettare il pattern, monitorare il sistema e effettuare il deploy del sistema.
- con la pratica diventa facile anche progettare e creare il sistema.
- permette di scorporare agevolmente problemi complessi e di garantir loro sufficiente spazio e potenza di esecuzione.

In conclusione si crede che nelle soluzioni moderne l'uso del pattern Serverless sia essenziale per le applicazioni che devono e vogliono garantire una forte disponibilità, corenza tra i dati e tolleranza di partizione.

References

[1] What serverless computing really means, and everything else you need to know

[2] AWS Lambda documentation

[3] AWS Cognito documentation

[4] AWS DynamoDB documentation

[5] AWS S3 documentation