PROUD2BE CLOUD!

Caching e long-jobs con notifiche in tempo reale in una Web App Serverless basata su AWS

9 Luglio 2021 - 7 min. read

AWS Lambda | Serverless

Le infrastrutture Serverless offrono enormi vantaggi rispetto alle infrastrutture server "classiche". Basti pensare ad un'applicazione Web serverless di base basata su AWS sviluppata utilizzando AWS Lambda come livello di backend, DynamoDB on-demand (database), Cognito per l'autenticazione e S3-CloudFront per il frontend. Un'applicazione come questa è perfettamente in grado di scalare automaticamente e in tempo reale, per accettare qualsiasi quantità di traffico. Rispetto ad una infrastruttura basata su EC2, avrà un costo notevolemnte minore pur essendo molto più semplice da implementare e mantenere. Adottare un paradigma Serverless sembra quindi una scelta quasi ovvia nella maggior parte delle situazioni: si andrà a spendere meno per un'applicazione in grado di scalare meglio, che avrà un maggiore isolamento, una migliore sicurezza e uno sforzo di manutenzione molto inferiore.

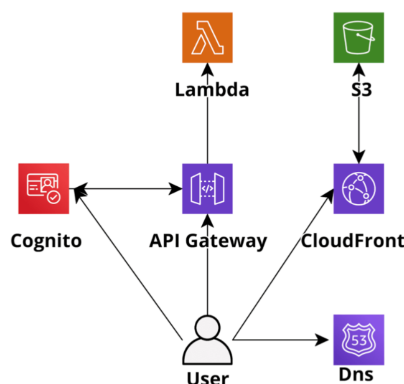


Figura 1: Esempio di applicazione serverless di base

Ad oggi sono stati creati diversi framework di sviluppo incentrati su AWS Lambda che rendono lo sviluppo di applicazioni serverless su AWS un gioco da ragazzi (ad es. Chalice per Python, Serverless-framework per diversi linguaggi). Inoltre, essendo dotati di tutte le necessarie integrazioni, è abbastanza semplice creare pipeline CI/CD utilizzando gli strumenti DevOps nativi di AWS (CodeBuild, CodeDeploy, CodePipeline) e integrare le applicazioni serverless con molti altri servizi AWS come SQS, SNS e Kinesis.

Tuttavia, il passaggio a un'architettura serverless comporta nuovi problemi e soluzioni rispetto alle architetture più tradizionali. La memorizzazione nella cache di risorse sia interne che esterne (es. query di database, risorse frontend ed API di terze parti) in particolare richiede strumenti e tecniche diverse da quelle utilizzate in un'architettura più tradizionale, così come anche la gestione dei processi di lunga durata.

Caching

In un'applicazione tradizionale (es. Ruby on Rails o Django) la cache è gestita con un deployment redis/memcached locale oppure centralizzato che viene spesso utilizzato per tutto, dai componenti frontend alle query DB e alle risposte alle chiamate API esterne.

In un ambiente serverless, la memorizzazione nella cache deve essere non solo molto veloce da accedere (pochi ms) e semplice da usare, ma anche senza necessità di connessioni persistenti e infinitamente scalabile. Questi requisiti rendono DynamoDB la scelta più ovvia e solitamente la migliore: quando si configura una tabella come on-demand, infatti, la sua capacità di scrittura e lettura si ridimensiona automaticamente consentendo all'applicazione di rimanere reattiva anche in caso di improvviso aumento del traffico e il tempo di accesso alla tabella da Lambda rimane nell'ordine dei millisecondi. Se è necessario un tempo di accesso ancora più basso è possibile attivare Dynamo DAX Accelerator che può abbassare ulteriormente la latenza di lettura passando così da millisecondi a microsecondi!

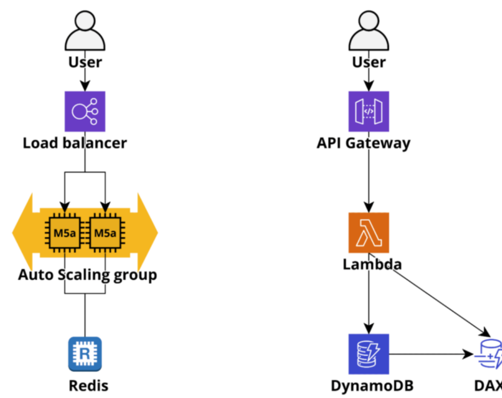


Figura 2: memorizzazione nella cache in AWS: app basata su EC2 a sinistra, serverless a destra

Jobs di lunga durata in un'applicazione Web serverless

In un'architettura tradizionale, un'applicazione Web di solito gestisce attività con tempi di esecuzione prolungati generando thread e inviando notifiche tramite connessioni websocket aperte. Un'applicazione basata su Lambda, invece, normalmente gestisce un'attività di lunga durata utilizzando AWS SQS o Kinesis come servizi di accodamento e container Fargate, oppure con Lambda autonome come worker. È anche possibile eseguire attività particolarmente complesse o con logica intricata avviando flussi di lavoro serverless con AWS StepFunctions.

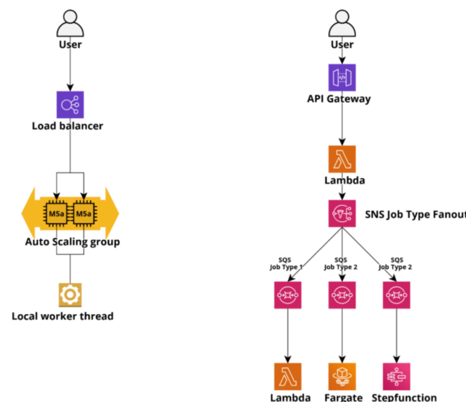


Figura 3: Job a lunga esecuzione in un'applicazione web AWS: app basata su EC2 a sinistra, serverless a destra

Infine, spesso è utile inviare una notifica direttamente al browser client per aggiornare lo stato dei componenti del frontend in risposta a eventi come come stato di esecuzione delle attività in esecuzione, notificare all'utente azioni eseguite da altri utenti (ad esempio nei giochi online), recapitare messaggi e sempre più spesso notificare agli utenti i cambiamenti di stato di dispositivi IoT.

Notifiche in tempo reale

Mentre nelle applicazioni classiche è possibile utilizzare direttamente i websocket, anche tramite AWS ELB che supportano HTTP/2, per le applicazioni serverless è necessario sfruttare il supporto Websocket di AWS ApiGateway che è anche supportato nativamente da diversi framework serverless, come Chalice. Quando una connessione WebSocket viene stabilita da un client, una lambda può essere invocata dall'hook *$connect* di ApiGateway e sarà in grado di registrare l'id di connessione a un database, solitamente DynamoDB. Quando un utente si disconnette, viene invocato *$disconnect* e ciò consente alla nostra applicazione di eliminare il record dalla tabella delle connessioni. Sviluppare una logica per inviare notifiche è piuttosto semplice: quando un messaggio deve essere consegnato dal backend a un utente, l'ApiGateway @connections POST Api viene invocato utilizzando gli id delle connessioni websocket aperte dell'utente e ApiGateway si occupa di inoltrare il messaggio nel canale websocket aperto.

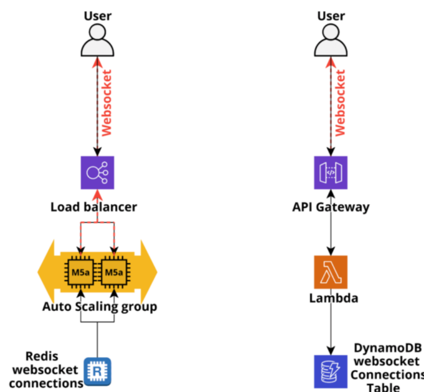


Figura 4: Notifiche Websocket in tempo reale in un'applicazione Web AWS: app basata su EC2 a sinistra, serverless a destra

Un esempio del mondo reale

Sebbene queste tecniche siano particolarmente utili per le applicazioni ad alto traffico, anche le app a basso traffico possono trarre grandi vantaggi dalla loro implementazione, in particolare quelle che gestiscono flussi di lavoro complessi.

La seguente architettura di esempio è una versione notevolmente semplificata dell'architettura che abbiamo effettivamente realizzato per un'applicazione che consente ad un cliente di gestire dinamicamente i bucket S3 e gli utenti IAM che vi accedono.

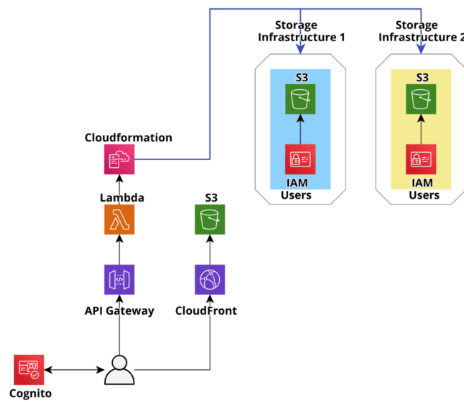


Figura 5: Prima versione dell'infrastruttura applicativa di esempio

L'applicazione consente ai project manager del cliente di creare facilmente bucket S3 protetti in account AWS dedicati, specifici per progetto, isolati e hardenizzati. Una volta creati i bucket, è anche possibile generare utenti IAM gestiti per accedervi. Le credenziali dell'utente IAM possono quindi essere condivise in modo sicuro con terze parti per consentire loro di scaricare e/o caricare file da S3 utilizzando sistemi legacy on-premise, sfruttando le API di S3 o SFTP (AWS Transfer for SFTP). Utenti e Bucket possono essere facilmente aggiunti e rimossi da ciascun progetto e le autorizzazioni utente possono essere gestite tramite una semplice interfaccia utente.

Per semplificare lo sviluppo del backend e allo stesso tempo rendere l'applicazione più resiliente, abbiamo deciso di non configurare alcun database e utilizzare semplicemente il backend per creare, aggiornare e modificare dei modelli di Cloudformation che descrivono l'intera infrastruttura AWS. In questo modo viene rafforzata la consistenza dell'applicazione: ogni azione eseguita viene automaticamente registrata e, in caso di errore durante la creazione di una risorsa, i rollback vengono eseguiti automaticamente direttamente dal servizio Cloudformation.

Tuttavia questo approccio presenta due svantaggi principali, uno per le operazioni LIST/GET e uno per le operazioni CREATE/UPDATE.

Infatti, ogni volta che un utente finale elenca le risorse esistenti, il backend deve recuperare tutti i modelli di CloudFormation, analizzarli per elencare le risorse e infine restituire la risposta. Questo flusso deve essere eseguito almeno su un modello ogni volta che un utente esegue un'operazione LIST o GET. Per account con dozzine di bucket e utenti ciò può richiedere diversi secondi, rendendo l'esperienza dell'utente molto scadente e potenzialmente, in alcuni casi estremi, può portare al superamento del limite di risposta di ApiGateway (30 secondi).

Il secondo problema riguarda le operazioni di aggiornamento e si manifesta quando sono connessi più clienti: se un gestore aggiorna un bucket o un utente, tutti gli altri gestori non potranno modificarlo fino al termine dell'esecuzione di Cloudformation, che in questo caso d'uso richiede solo pochi secondi. Tuttavia, gli altri utenti connessi non hanno modo di sapere quando un utente viene aggiornato e potrebbero tentare di modificarlo causando errori imprevisti che, sebbene innocui, rendono l'esperienza dell'utente poco fluida.

Oltre a questi problemi c'è anche un'ulteriore potenziale inconveniente: poiché l'infrastruttura descritta è "GET heavy" verso le API di CloudFormation, se un numero sufficientemente grande di gestori accede, il rate limit dell'API di CloudFormation potrebbe potenzialmente essere superato con conseguenti ulteriori rallentamenti (gli SDK AWS implementano l'esponential backoff).

Per risolvere tutti questi problemi, dopo l'MVP iniziale, abbiamo applicato il design pattern sopra descritto per rendere applicazione pronta per essere messa in produzione: abbiamo utilizzato DynamoDB per memorizzare nella cache lo stato di CloudFormation e abbiamo utilizzato l'integrazione di CloudFormation con SNS per aggiornare lo stato in tempo reale tramite connessioni Websocket (gestite da Api Gateway) verso i client degli utenti e allo stesso tempo per aggiornare la cache su dynamoDB con l'ultimo stato di Cloudformation, in modo che lo stato nel DB di cache rispecchi sempre lo stato di Cloudformation.

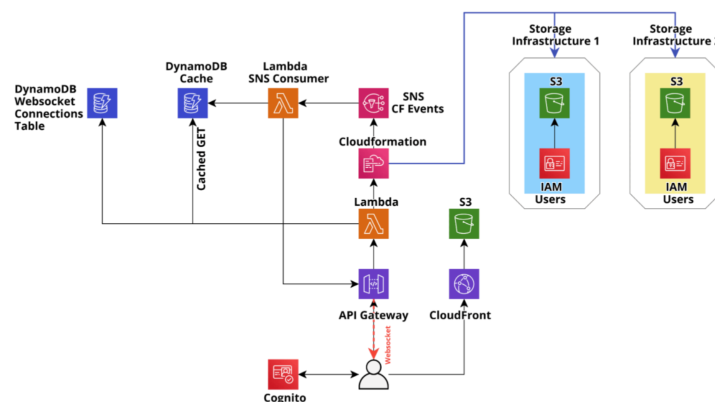


Figura 6: versione finale dell'infrastruttura applicativa di esempio

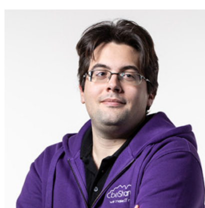
Questi accorgimenti hanno prodotto enormi miglioramenti nei tempi di risposta delle APIs del backend, che per le richieste GET/LIST sono scesi costantemente al di sotto dei 200 ms. Inoltre, il tempo di attesa degli utenti dopo il lancio di un aggiornamento

di CloudFormation risulta notevolmente diminuito e lo stato delle risorse è coerente su tutti i client connessi.

Il cliente è stato più che soddisfatto del risultato finale e un cliente felice è sempre la migliore conferma di un lavoro ingegneristico ben fatto!

Se hai domande sui modelli di progettazione per DynamoDB o su qualsiasi altro argomento riguardante (o non riguardante) questo articolo, non esitare a contattarci! Non vediamo l'ora di discuterne :)

Ci vediamo qui, su **#Proud2beCloud** tra 14 giorni per un nuovo articolo!



Matteo Moroni

DevOps e Solution Architect di beSharp, mi occupo di sviluppare soluzioni Saas, Data Analysis, HPC e di progettare architetture non convenzionali a complessità divergente. Appassionato di informatica e fisica, da sempre lavoro nella prima e ho un PhD nella seconda. Parlare di tutto ciò che è tecnico e nerd mi rende felice!

Copyright © 2011-2021 by beSharp srl - P.IVA IT02415160189