Final Report

Andrea Rettaroli
andrea.rettaroli@studio.unibo.it

Luglio 2021

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

Contents

1 Obiettivi

1.1 Definizione degli obbietivi

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

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

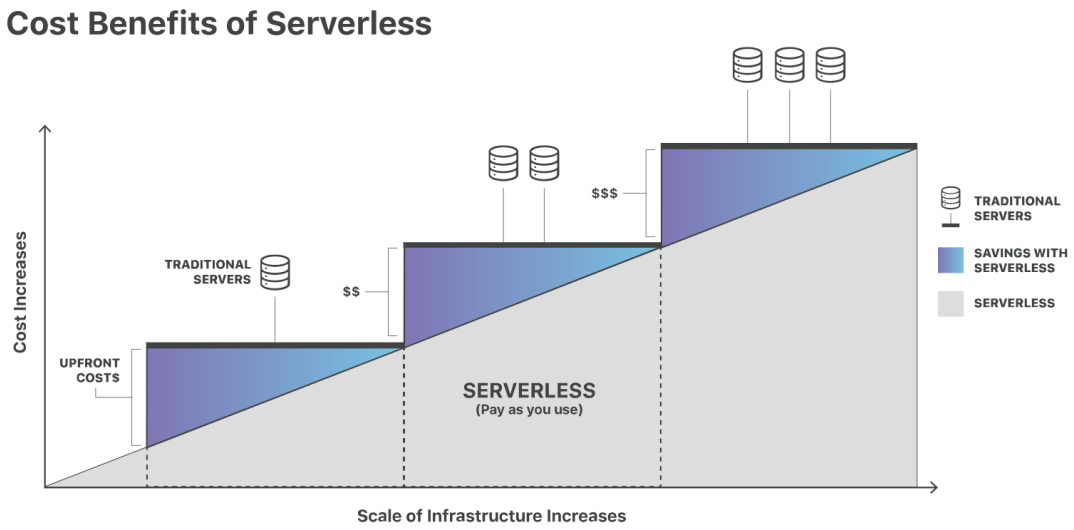
2 Architetture e paradigma Serverless

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

2.1 Serverless Definizione

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

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



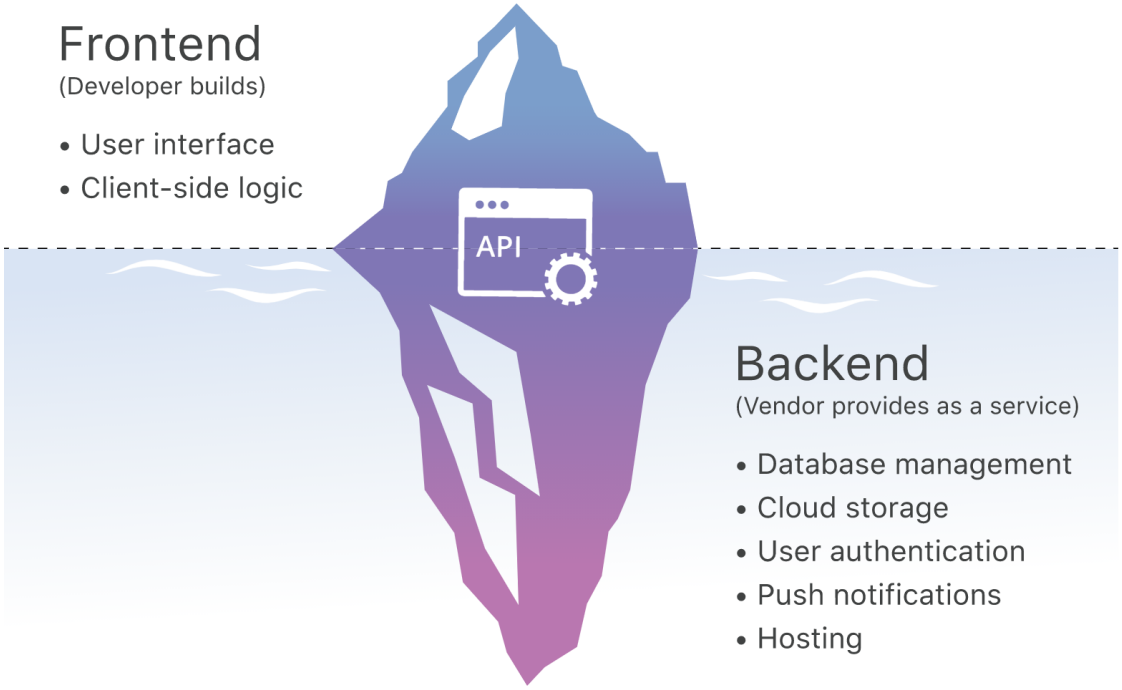
Dal grafico è evidente che il costo iniziale è maggiore, ma se paragonato al costo dell'allestimento di un infrastruttura server fisica interna o in cloud è inferiore e questo trend rimane anche al crescere della dimensione dell'infrastruttura. La differenza sta nel pagare le risorse in base all'elaborazione.

2.2 Architetture Serverless

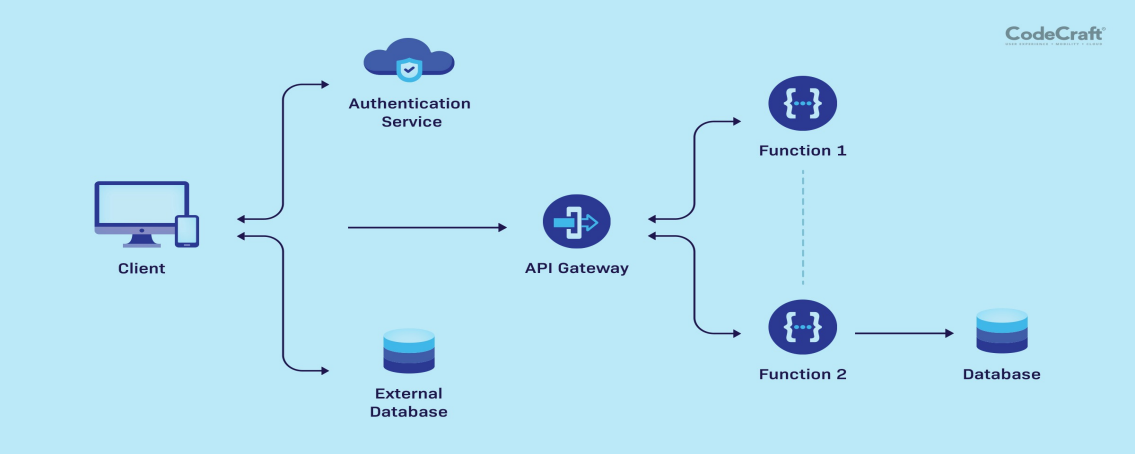
Ora che abbiamo chiarito il concetto di Serverless possiamo analizzare meglio le tipiche architetture che supportano questo paradigma e che ne permettono il funzionamento, generando quel livello in più di astrazione.

2.3 Backend as a Service

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



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



2.4 Functions as a Service

2.5 Vantaggi e svantaggi

//spostare prima della presentazione dei progetti.

3 Requisiti

3.1 Requisiti funzionali

Per i requisiti funzionali variano a seconda degli esempi pratici che verranno prodotti. Iniziamo a fare una distinzione dei requisiti funzionali in un'architettura Backend as a Service e in un'architettura Functions as a Service che descriveremo nello specifico in seguito. In generale per la produzione di parte di questi progetti è necessario essere in possesso di credenziali AWS e di un ruolo. Inoltre nell'approccio Serverless-Functions as a Service, che illustremo in seguito, la macchina dove è in esecuzione il server necessita di interfacciarsi con i servizi AWS e quindi di conoscere le credenziali e avere un ruolo.

3.2 Requisiti funzionali Baas

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

3.3 Requisiti funzionali Faas

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

3.4 Requisiti non funzionali

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

3.5 Scenarios

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

3.6 Self-assessment policy

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

4 Requirements Analysis

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

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

5 Design

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

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

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

5.1 Structure

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

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

5.2 Behaviour

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

5.3 Interaction

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

6 Implementation Details

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

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

7 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.

8 Deployment Instructions

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

configuration step.

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

9 Usage Examples

Show how to use the produced software artefacts.

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

10 Conclusions

Recap what you did

10.1 Future Works

Recap what you did *not*

10.2 What did we learned

Racap what did you learned

Stylistic Notes

Use a uniform style, especially when writing formal stuff: X, X, $\mathbf{X}$, $\mathcal{X}$, $\mathtt{X}$ 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.