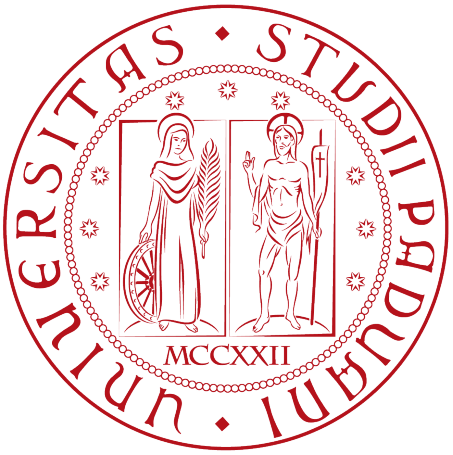
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Generazione automatica di infrastrutture cloud da immagini vettoriali

Tesi di laurea

Relatore
Prof. Davide Bresolin

Laureando
Alessandro Baldissera
Matricola 1216742

Anno Accademico 2022-2023

Non ho alcun talento particolare. Sono solo appassionatamente curioso.

— Albert Einstein

Dedicato alla mia famiglia e ai miei amici

Sommario

Questa tesi descrive la progettazione e la realizzazione di un software per la generazione automatica di template di infrastruttura cloud Amazon Web Services. Il lavoro è stato svolto durante un tirocinio presso l'azienda zero12 s.r.l., dove ho sviluppato un insieme di servizi web per convertire un'immagine vettoriale in un template CDK per la gestione dell'infrastruttura via codice. La piattaforma che ho sviluppato comprende un protocollo per la creazione del disegno, l'architettura dei micro servizi per la conversione e un'interfaccia utente per l'utilizzo dei servizi. Per la realizzazione dei servizi AWS ho utilizzato il linguaggio Typescript e il framework Serverless. Per l'interfaccia utente ho utilizzato il linguaggio Typescript e il framework React.

Abstract

This thesis describes the design and implementation of a software that automatically generate an Amazon Web Services infrastructure template from a vectorial image. The work was done at zero12 s.r.l. where I developed a set of web services to convert a vectorial image to a CDK template to manage infrastructure as code. The developed platform includes a drawing protocol, the micro-service architecture to perform conversion and an user interface. I used Typescript language and Serverless framework to implements the services. I used Typescript language and React framework for the user interface.

"Quiet people have the loudest minds."

— Stephen Hawking

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Davide Bresolin, relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura del lavoro.

Desidero ringraziare con affetto i miei genitori per il sostegno, il grande aiuto e per essermi stati vicini in ogni momento durante gli anni di studio.

Ho desiderio di ringraziare poi i miei amici per tutti i bellissimi anni passati insieme e le mille avventure vissute.

Desidero ringraziare i miei compagni di corso per aver condiviso assieme gli anni di università.

Ringrazio "gli amici di matematica" per avermi accompagnato in un corso di studi non adatto a me e per essermi stati vicini anche nella scelta di terminare quel percorso.

Infine un ringraziamento anche ai colleghi di zero12 per questi due mesi di meraviglioso tirocinio.

Padova, Settembre 2023 Alessandro Baldissera

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei listati

Capitolo 1

Introduzione

In questo capitolo vengono introdotte l'azienda e il progetto di stage.

1.1 L'azienda

zero12 s.r.l è una *startup* nata nel 2012 e specializzata nello sviluppo di applicazioni su *cloud Amazon Web Services*, applicazioni *web* e *mobile*. Il team di zero12 cura la progettazione e lo sviluppo delle applicazioni e dei servizi in ogni dettaglio offrendo al cliente soluzioni flessibili e scalabili. zero12 s.r.l non si occupa solo di sviluppo di applicazioni ma crea assieme al cliente un percorso di innovazione per integrare le tecnologie più adatte alle loro necessità per migliorare processi produttivi o il modo in cui si presentano nel mondo web, con siti *web* originali e innovativi.

1.2 L'idea

Per creare un'applicazione su cloud si fanno interagire tra di loro diversi servizi offerti dal *cloud provider*. I servizi necessari e come interagiscono tra di loro sono descritti in un file di configurazione. Il file di configurazione può essere difficile da interpretare: per avere una vista più chiara di quello che si sta creando si può utilizzare uno schema grafico per visualizzare l'infrastruttura. Da qui si vuole creare un'applicazione che partendo da uno schema di infrastruttura creato tramite Draw.io, generi un'applicazione *AWS Cloud Development Kit* che si occupa di creare le istanze dei servizi necessari automaticamente. Un'applicazione *CDK* si scrive con il linguaggio *Typescript*, ed essa porta con sè tutti i vantaggi di della gestione dell'infrastruttura cloud tramite codice, come la possibilità di versionamento e approvazione delle modifiche. Questo approccio inoltre permette di gestire le modifiche dell'infrastruttura solo nel disegno, senza preoccuparsi di dover mantenere sincronizzate a mano la configurazione e lo schema.

1.3 Organizzazione del testo

Il secondo capitolo introduce al progetto di stage mettendo in evidenza obiettivi formativi e pianificazione del lavoro;

Il terzo capitolo approfondisce gli aspetti principali legati alla gestione infrastrutturale nel *cloud*;

Il quarto capitolo approfondisce gli aspetti principali della libreria Cloud Development Kit (CDK) di AWS;

Il quinto capitolo mostra una panoramica sulle tecnologie coinvolte nello sviluppo del progetto;

Il sesto capitolo approfondisce gli aspetti legati alla progettazione delle diverse parti che compongono il *software*;

Nel settimo capitolo si descrive l'implementazione del progetto;

Nell'ottavo capitolo si discutono delle considerazioni sull'uso effettivo del prodotto;

Nel nono capitolo sono descritte le conclusioni dell'esperienza dello stage e degli obiettivi raggiunti.

1.4 Guida alla lettura

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

- gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
- i termini in lingua straniera sono evidenziati con il carattere *corsivo*.

Capitolo 2

Descrizione

Nel seguente capitolo verrà fatta una una breve introduzione al progetto, concentrandosi sugli obiettivi formativi e la pianificazione delle attività.

2.1 Introduzione al progetto

L'azienda da qualche tempo ha introdotto l'uso del paradigma Infrastructure as Code che si pone come obiettivo l'uso di codice per eseguire il *deploy* di un'applicazione in ambito cloud. Questo paradigma però introduce la necessità di creare delle rappresentazioni schematiche dell'applicazione, in quanto non tutte le persone incluse nello sviluppo di un progetto software hanno le capacità per leggere e capire del codice sorgente. Questo però introduce la necessità di mantenere sincronizzati codice sorgente e disegno. Per questa ragione si è pensato di integrare le configurazioni volute direttamente nel disegno per poi convertirle in automatico in nel codice sorgente che si occuperà di effettuare il *deploy* su AWS.

2.2 Obiettivi formativi

I principali obiettivi formativi di questo progetto sono:

- Comprendere come l'infrastruttura *cloud* di un'applicazione possa essere gestita tramite codice;
- Creare un'API HTTP RESTful;
- Comprendere e utilizzare le principali tecnologie *cloud* AWS.

2.3 Obiettivi del progetto

Durante il primo giorno di stage si è svolto un incontro con il tutor aziendale ed altri membri del *team*, esperti di infrastruttura, per definire gli obiettivi del progetto. Essendo un progetto sperimentale, non sono stati definiti dei requisiti formalmente. I principali obiettivi del progetto sono:

- Definire uno *standard* di disegno sulla piattaforma Draw.io;
- Definire un insieme di micro-servizi per effettuare la conversione da disegno a codice;
- Poter convertire correttamente una rappresentazione di un VPC;
- Poter convertire una istanza di EC2
- Poter convertire una funzione Label;
- Poter convertire un bucket S3;
- Poter convertire una istanza di DynamoDB;
- Poter convertire una istanza di Api Gateway con la possibilità di integrare in esso le funzioni Lambda;
- Creare un'interfaccia per l'utente.

2.4 Pianificazione

La durata complessiva dello stage è stata di 8 settimane per un totale di 320 ore. Secondo il piano di lavoro definito ad inizio stage con l'azienda le attività sono state distribuite come riportato nella tabella 2.1.

Periodo	Durata (ore)	Attività
1	40	Studio delle tecnologie AWS, modalità di lavoro, introduzione al linguaggio di programmazione e organizzazione del *software* a micro-servizi
2	80	Studio e disegno di architetture *cloud* tramite Draw.io e studio del formato del file di esportazione
3	80	Progettazione e sviluppo dell'algoritmo che dato il disegno in *input* genera il template CDK e sviluppo API

Periodo	Durata (ore)	Attività
4	80	Creazione dell'infrastruttura tramite Draw.io per la generazione dell'architettura del *software* e *deploy* tramite i servizi di *CI/CD*
5	40	*Test* e scrittura documentazione di quanto sviluppato

Tabella 2.1: Tabella riassuntiva della pianificazione delle attività di stage.

Durante lo svolgimento dello stage il piano di lavoro ha subito delle modifiche e di conseguenza il consuntivo nella sezione conclusiva diverge dalla pianificazione. Durante tutta la durata dello stage sono stati effettuati degli *stand-up* giornalieri con il tutor aziendale per monitorare lo stato di avanzamento del progetto e discutere dei problemi sorti. Alla fine dello sviluppo si è tenuta una presentazione interna del prodotto.

Capitolo 3

Gestione dell'infrastruttura

In questo capitolo verranno descritte le principali modalità di gestione delle infrastrutture cloud, sia con approcci indipendenti dalla piattaforma, sia con approcci dedicati ad uno specifico cloud provider.

3.1 Scopo

In ambito *cloud* le applicazioni sono create utilizzando diversi servizi offerti dal *provider*. Cercando l'efficienza e la velocità nei *software* che si vogliono implementare, nel tempo sono nati decine di servizi diversi e sempre più specializzati su degli *use case* specifici. Per gestione dell'infrastruttura si intende l'installazione e la configurazione dei servizi *cloud*, ma anche monitoraggio, ottimizzazione e manutenzione dell'infrastruttura stessa. Nelle sezioni successive si vedrà com'è implementata attualmente la gestione dell'infrastruttura, focalizzandosi sugli aspetti di installazione, configurazione e manutenzione.

3.2 Gestione manuale dell'infrastruttura.

L'approccio più semplice per la gestione dell'infrastruttura *cloud* di un'applicazione consiste nello istanziare e configurare le risorse e servizi necessari manualmente dalla console di amministrazione. Un approccio di questo tipo facilita lo sviluppo di nuove applicazioni in quanto non richiede l'integrazione di strumenti e tecnologie terze per la gestione dell'infrastruttura. D'altro canto, gestire manualmente l'infrastruttura ha delle forti controindicazioni.

Dispersione dei servizi La console di amministrazione di un *cloud provider* presenta sempre delle sezioni dedicate e isolate per ogni servizio offerto. Questo non permette di

aver riunito in un uno spazio tutti i servizi e risorse di un progetto, non permettendo agli sviluppatori di avere una visione generale dell'infrastruttura.

Tracciamento impossibile È impossibile tenere traccia dei cambiamenti effettuati all'infrastruttura riguardo ai servizi, le configurazioni e chi ha effettuato la modifica. Questo limite non permette di effettuare il *rollback* in caso di errori ed rende impossibile tracciare il momento esatto nel quale un problema è sorto.

Difficoltà di manutenzione Il personale tecnico cambia continuamente e questo rende molto difficile la manutenzione di un'infrastruttura nel lungo periodo in quanto è difficile riuscire a tramandare tutte le indicazioni necessarie per poter operare.

3.3 Gestione dell'infrastruttura tramite codice

Per ovviare ai problemi indicati precedentemente, si è introdotto il paradigma *Infrastructure as Code* che prevede la gestione dell'infrastruttura tramite file di configurazione o codice sorgente. Questo approccio prevede la creazione di file di configurazione o di un'applicazione che si occupi di istanziare e configurare tutti i servizi necessari. In generale si risolvono tutti problemi citati nella gestione manuale in quanto si centralizzano in un unico punto tutte le definizioni di servizi e risorse, e si può introdurre il versionamento con strumenti come Git che permettono di tracciare le modifiche ed eventualmente fare il *rollback* ad una versione precedente della configurazione. Questo approccio richiede la conoscenza più o meno approfondita di un qualche linguaggio di programmazione o del formato di un file di configurazione.

3.3.1 Gestione dell'infrastruttura tramite file di configurazione

L'approccio più semplice per la gestione dell'infrastruttura *cloud* prevede la scrittura di un file di configurazione per la definizione delle risorse e dei servizi necessari. Ci sono diversi servizi che implementano questo approccio e possono essere sia proprietari di un *cloud provider* come AWS CloudFormation oppure possono essere agnostici come Terraform. L'uso di file di configurazione è la via più semplice per applicare il paradigma *Infrastructure as Code* che si pone come obiettivo principale la risoluzione dei problemi introdotti dalla gestione manuale dell'infrastruttura. Infatti un file di configurazione unificato permette di centralizzare i servizi necessari all'applicazione e, integrando strumenti come Git per il versionamento, si possono tracciare anche le modifiche effettuate nel tempo.

3.3.2 Gestione dell'infrastruttura tramite codice eseguibile

Un secondo metodo per implementare il paradigma *Infrastructure as Code*, prevede di utilizzare un programma eseguibile per la gestione dell'infrastruttura. Un approccio come questo permette di sfruttare tutte le potenzialità di un linguaggio di programmazione per creare un'infrastruttura. Ad esempio si possono utilizzare i cicli per creare molte istanze della stessa risorsa, oppure creare istanze condizionali di servizi. Tendenzialmente le librerie che implementano questo approccio si basano sulla creazione di file di configurazione a seguito dell'esecuzione del codice. Ad esempio AWS Cloud Development Kit (CDK), è una libreria che permette di scrivere in diversi linguaggi di programmazione la propria infrastruttura *cloud*. L'esecuzione di un'applicazione CDK produce come output un file di configurazione che verrà usato dal servizio AWS CloudFormation, citato nella sezione 3.3.1.

3.4 Gli schemi infrastrutturali

I motivi per cui si possono voler realizzare degli schemi di un'infrastruttura sono molteplici. In primo luogo uno schema è tendenzialmente facile e veloce da capire e permette di visualizzare facilmente le relazioni tra i vari servizi. Questo permette di capire la struttura di un'applicazione in modo facile e veloce, permettendo a nuovi membri del *team* di sviluppo di iniziare a lavorare nel progetto nel minor tempo possibile. Inoltre in un progetto non lavorano mai solo tecnici e programmatori, ma ci sono molte altre figure che non hanno conoscenze informatiche sufficientemente approfondite per comprendere file di configurazione e codice eseguibile; da qui la necessità di utilizzare schemi. Per il *team* di sviluppo, o per la figura che si occupa della realizzazione dell'infrastruttura, però dover gestire sia lo schema che la configurazione su codice obbliga lo svolgimento di un doppio lavoro. In particolare in progetti nei quali l'infrastruttura varia spesso o in modo consistente, mantenere lo schema aggiornato diventa estremamente oneroso.

Capitolo 4

AWS Cloud Development Kit

In questo capitolo introdurrò i concetti fondamentali per l'uso della libreria AWS Cloud Development Kit

4.1 Introduzione a AWS Cloud Development Kit

Cloud Development Kit (CDK) è un *framework* per la definizione delle infrastrutture *cloud* su AWS. CDK fornisce una CLI con la quale è possibile generare un progetto CDK ed effettuare la successiva installazione dell'applicazione. CDK fornisce una libreria per la creazione di tutti i servizi disponibili su AWS sia a basso livello per avere una maggiore flessibilità che ad alto livello per ridurre la quantità di codice da scrivere. Permette inoltre di personalizzare i servizi in ogni parte. Implementando l'infrastruttura attraverso codice eseguibile è possibile sfruttare tutte le potenzialità offerte come cicli e condizioni per mutare a piacere l'infrastruttura che verrà installata su *cloud*.

4.2 Organizzazione

CDK organizza l'infrastruttura in costrutti e *stack*. Un costrutto rappresenta una risorsa mentre uno *stack* rappresenta un insieme di risorse.

4.2.1 I costrutti

I costrutti rappresentano i vari servizi offerti da AWS. Sono organizzati su 3 livelli:

Livello 1. Rappresenta il servizio a livello più basso, è personalizzabile in ogni parte ma richiede la scrittura di molto codice;

Livello 2. Rappresenta un servizio a livello più alto. Normalmente permette di personalizzare meno aspetti del servizio lasciando a valori di *default* le impostazioni meno utilizzate. Questi costrutti sono quelli più utilizzati nelle applicazioni CDK;

Livello 3. Rappresenta un servizio personalizzato dall'utente che può integrare anche della logica personalizzata. Normalmente questo livello è utilizzato per creare un insieme di servizi con delle impostazioni utilizzate spesso dall'utente.

Nell'ambito di questo progetto ho utilizzato solo costrutti di secondo livello e solo dove necessario i costrutti di primo livello. Al contrario non ho trovato necessario l'uso di costrutti di terzo livello.

4.2.2 Gli Stack

Gli *stack* sono dei contenitori di costrutti. Possono essere completamente isolati o condividere le risorse in essi istanziate con altri *stack*. In generale uno *stack* può essere organizzato per tipo di risorsa, ad esempio risorse computazionali, storage o database. Oppure possono essere organizzati per funzionalità dell'applicativo implementato. Nell'ambito di questo progetto ho seguito gli standard aziendali, raggruppando per tipo i servizi istanziati e condividendo le risorse negli *stack* dove necessario.

4.2.3 Entry Point

Con *entry point* si intende il file principale che esegue tutta l'applicazione CDK. Per quanto la CLI supporti il *deploy* di un singolo *stack*, è comunque possibile scrivere un file principale che istanzia tutti gli *stack* presenti. Nell'ambito del progetto ho scelto di creare un *entry point*, cioè un file che si occupa di creare un'istanza di ogni singolo *stack* definito nell'applicazione. Tramite l'*entry point* è dunque possibile eseguire integralmente l'applicazione CDK e quindi fare il *deploy* dell'infrastruttura tramite un singolo comando.

4.2.4 Configurazione dei servizi

La configurazione dei servizi può essere inserita direttamente nel codice sorgente oppure in un file di configurazione separato. L'azienda richiede di poter definire più set di configurazioni, ad esempio una configurazione per lo sviluppo e una per la produzione. Per questa ragione, dove possibile, si è preferito inserire le configurazioni in un file separato, in questo caso un file in formato JSON, ed aggiungere all'applicazione una funzione di lettura del file di configurazione. Alcune configurazioni non possono essere definite solo con un file di configurazione ma necessitano di essere scritte direttamente nel codice eseguibile.

Capitolo 5

Tecnologie e strumenti

In questo capitolo elencherò le tecnologie e gli strumenti coinvolti nello sviluppo del progetto

5.1 Tecnologie per il backend

Di seguito elencherò le tecnologie coinvolte nello sviluppo del *backend*.

5.1.1 Serverless Framework

Si tratta di un *framework* per lo sviluppo di API su infrastruttura AWS che permette di definire funzioni Lambda con i relativi *trigger* attraverso un semplice file di configurazione in formato YAML. Inoltre permette di eseguire il *deploy* nel *cloud* tramite un semplice comando, configurando l'architettura e caricando il codice eseguibile in automatico. Supporta inoltre dei plugin che possono estendere le funzionalità oppure aggiungere il supporto per altri servizi AWS.

5.1.2 Node.js

Node.js è un ambiente di *runtime* per il linguaggio Javascript basato su V8, la Javascript VM sviluppata da Google. Node.js fornisce una libreria *standard* che comprende vari moduli per le operazioni di base. Tramite il *packet manager* NPM si possono installare librerie e *framework* di terze parti per estendere le funzionalità.

5.1.3 Typescript

Overlay di Javascript che introduce la tipizzazione, OOP e *generics*. Introduce il concetto di *type-safety* a discapito di una fase di compilazione iniziale del codice.

5.1.4 AWS Lambda

Lambda è un servizio computazionale che permette di eseguire del codice senza la necessità di dover possedere o gestire dei *server*. Le Lambda inoltre scalano automaticamente in base al carico corrente. Tutti i servizi di *logging* e monitoraggio sono forniti in automatico da AWS. In generale le Lambda sono funzioni che seguono il *single responsibility principle*, quindi sono tendenzialmente di piccole dimensioni. Questo permette alle Lambda di essere caricate dal servizio di computazione ed eseguite in tempi brevi.

5.1.5 AWS Step Function

Il servizio Step Function è un orchestratore di servizi. Permette di creare dei *workflow* che integrano la maggior parte dei servizi AWS e di eseguire codice senza doversi preoccupare della gestione della concorrenza. Il *workflow* è implementato tramite una macchina a stati, il cui flusso di esecuzione si basa sugli eventi. Le attività della macchina sono chiamate *task* e non prevedono solo l'esecuzione di codice di *business logic*, ma possono anche interfacciarsi con altri servizi.

5.1.6 AWS DynamoDB

Si tratta di un *document database* NoSQL e scalabile automaticamente in base al traffico. Supporta lettura e scrittura di dati di qualsiasi dimensione senza degradamento delle performance generali del servizio. Inoltre sono supportati *backup on-demand* e *backup on-time* per un ripristino efficace dei dati in caso di corruzione dei dati oppure di operazioni errate sulla tabella.

5.1.7 AWS S3

Si tratta di un servizio di *object storage* scalabile che supporta la crittografia. è un servizio molto versatile per la gestione dei *file*, supporta operazioni di elaborazione sugli oggetti da parte di altri servizi, ma può anche gestire *backup* e servire siti *web* statici. S3 si integra perfettamente con gli altri servizi AWS, permettendo di avviare *workflow* automaticamente.

5.1.8 AWS Api Gateway

Api Gateway è un servizio che permette di creare delle API RESTful, basate sullo *standard* HTTP, oppure con WebSocket. Offre scalabilità, sicurezza e monitoraggio *out-of-the-box*. Il servizio può essere integrato nell'ecosistema AWS automaticamente per offrire autenticazione degli utenti, *trigger* di eventi ed esecuzione di operazioni *standard* su altri servizi senza la necessità di scrivere codice.

5.1.9 AWS Cognito

AWS Cognito è un *identity provider* che fornisce autenticazione e autorizzazione per siti *web* e applicazioni *mobile*. Gestisce automaticamente tutti gli aspetti legati alla sicurezza. Può inoltre integrarsi perfettamente con altri *identity provider* basati su protocolli come OAuth, SAML e OpenID.

5.2 Tecnologie per il frontend

Di seguito elencherò le tecnologie coinvolte nello sviluppo del *frontend*. Tra le tecnologie già citate ho utilizzato anche Typescript.

5.2.1 React

React è un *framework web open-source* per la realizzazione di *single-page application*. È attualmente mantenuto da Meta e da una comunità di aziende e sviluppatori indipendenti. React implementa solo il *rendering* del DOM mentre si basa su librerie di terze parti per la gestione di tutti gli altri aspetti di una *single-page application*.

5.2.2 Amplify

Amplify è una libreria fornita da AWS che si interfaccia con AWS Cognito. Oltre alle funzionalità di autenticazione fornisce anche delle componenti grafiche che possono essere integrate nell'applicazione.

5.3 Strumenti a supporto dello sviluppo

Di seguito elencherò i principali strumenti utilizzati a supporto dello sviluppo del prodotto.

5.3.1 Git

Git è un popolare strumento di versionamento *open-source* del codice sorgente che basa il suo funzionamento sul salvare le differenze tra una versione e la successiva dei file. Supporta anche il *branching* per poter lavorare indipendentemente su più rami di sviluppo; inoltre, dove possibile, gestisce automaticamente l'unione delle modifiche da rami diversi.

5.3.2 Microsoft Visual Studio Code

Visual Studio Code è un *editor* di testo sviluppato da Microsoft, che grazie al suo ecosistema di estensioni permette di essere utilizzato con molti linguaggi diversi, e può anche integrarsi con diversi sistemi di *debug* del codice.

5.3.3 AWS CodeCommit

AWS CodeCommit è il servizio di gestione del codice sorgente basato su Git e fornito da AWS. Permette inoltre di integrarsi facilmente con gli altri servizi di AWS.

5.3.4 AWS CodeBuild

AWS CodeBuild è il servizio di CI/CD offerto da AWS, si integra con CodeCommit e permette di eseguire qualsiasi operazione allo scatenarsi di eventi.

5.3.5 Postman

Postman è un *software* che permette di testare le API RESTful HTTP e WebSocket. Permette di personalizzare le richieste in ogni singola parte, dagli *header* al *body* e supporta tutti i metodi HTTP *standard*.

Capitolo 6

Progettazione

In questo capitolo descriverò la fase di progettazione del prodotto.

6.1 Flusso di esecuzione ad alto livello

Ho iniziato la progettazione pensando ad una strategia per risolvere il problema ad alto livello. Ho deciso di adottare una strategia simile a quella utilizzata nei compilatori convenzionali. Il flusso che ho pensato è schematizzato nella figura 6.1. La prima fase si occupa di leggere il file e di estrarne tutti i singoli blocchi del disegno e le relative informazioni necessarie a completare l'elaborazione successiva, ad esempio vengono estratti tutti i dati impostati dall'utente e la coordinate. Successivamente si analizzano queste informazioni, validandole e andando a creare una struttura dati che rispecchi l'architettura rappresentata. Infine si interpreta tale struttura dati e si genera il codice finale.

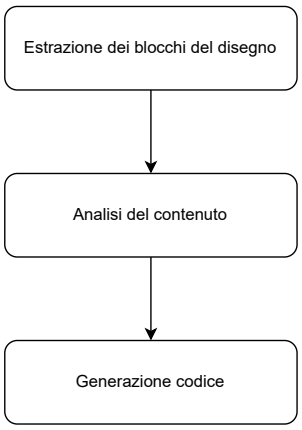


Figura 6.1: Diagramma di sequenza del processo di conversione.

6.2 Progettazione delle funzioni Lambda

Per *standard* aziendale le funzioni Lambda devono seguire il *single responsibility principle*, per questo motivo rispetto alla progettazione ad alto livello ho scelto di implementare le una Lambda per la prima fase di esecuzione, una lambda per la seconda e infine una lambda per ogni file di codice sorgente da generare. Ho sviluppato anche altre funzioni Lambda per implementare anche altre funzioni, quali le comunicazioni WebSocket e l'autenticazione dell'utente.

6.3 Progettazione della Step Function

Analizzando i possibili modi per interfacciare un *client* con il *backend* mi sono reso conto che non posso affidare ad esso la chiamata alle singole funzioni di elaborazione. Questo perché non posso garantire che il *client* richiami nel giusto ordine le funzioni, e che non manometta i dati scambiati. Per questa ragione ho deciso di integrare una Step Function che orchestri le chiamate alle singole funzioni. Questo mi permette di risolvere non solo i problemi citati prima ma anche ispezionare i dati di ritorno dalle singole funzioni e prendere delle decisioni basandomi su tali informazioni. Infine con le Step Function posso sfruttare anche il parallelismo, dove possibile, per aumentare le prestazioni. Nella figura 6.2 riporto la macchina a stati che rappresenta il flusso di esecuzione della Step Function.

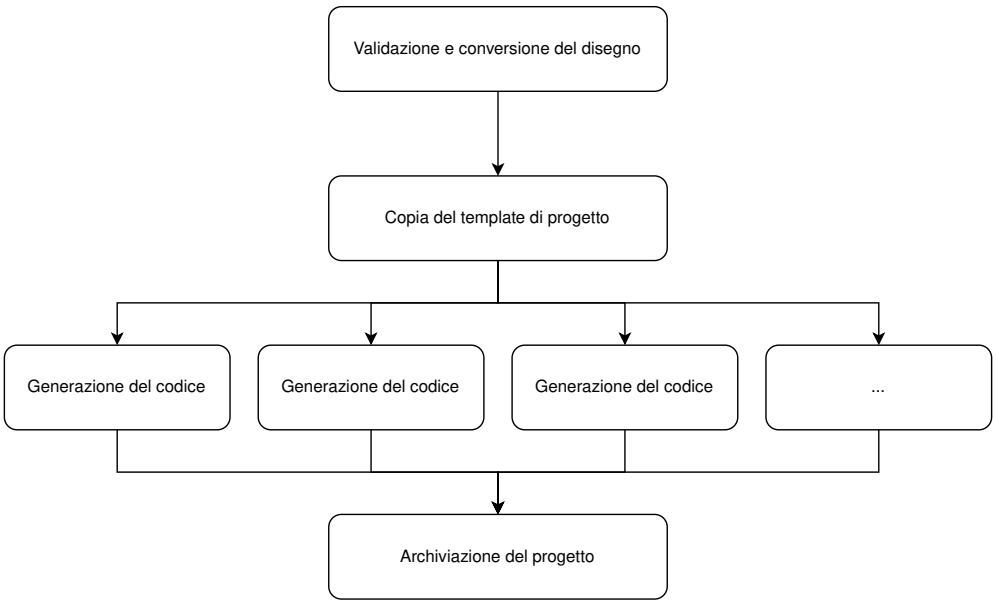


Figura 6.2: Diagramma del flusso di esecuzione della Step Function

All'inizio vengono eseguite le due Lambda in serie, successivamente per la generazio-

ne del codice, non avendo delle dipendenze tra le funzioni, ho deciso di eseguire tutto il workflow in parallelo per ottenere maggiori performance. Le Lambda che generano gli *stack* ho scelto di eseguirle condizionalmente (vedi figura 6.3), facendo controllare alla Step Function i dati di *input* della Lambda per determinare se lo *stack* che genera esiste. Nel caso in cui lo *stack* non esiste si salta l'esecuzione, questo permette di evitare l'esecuzione di funzioni che non genererebbero un risultato, oltre a risparmiare dal punto di vista economico sull'esecuzione della funzione.

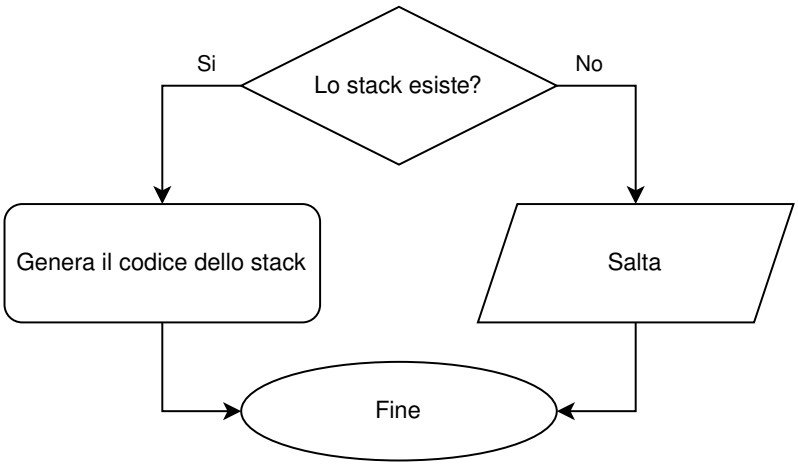


Figura 6.3: Diagramma dell'esecuzione condizionale della generazione del codice

Nello specifico il controllo sull'esistenza di uno *stack* viene eseguito su una struttura dati, di cui una piccola parte viene riportata nel listato 6.1. La prima funzione della Step Function che si occupa di convertire le informazioni del disegno, raggruppa i blocchi in liste omogenee di oggetti dello stesso tipo, non raggruppate per *stack*, e imposta un *flag* sull'esistenza dello *stack* stesso. Questo *flag* è necessario in quanto le tecniche di ispezione dei dati in *input* nei vari stati della Step Function sono molto limitate e non premettono, ad esempio, di ispezionare delle liste. Il raggruppamento per *stack* viene fatto direttamente nella funzione che si occupa di generare il codice.

```
1 {
2     "ComputeStack": true|false,
3     "ec2": [],
4     "lambda": [],
5 }
```

Listato 6.1: Struttura dati che raggurppa le informazioni del disegno. Parte relativa allo stack chiamato ComputeStack.

6.4 Progettazione delle API

Viste le considerazioni fatte nelle sezioni precedenti, le API risultanti sono così composte:

- Protocollo WebSocket: con il quale si richiede un URL per il caricamento del file e si ricevono le notifiche;
- API HTTP RESTful: con le quali otteniamo la lista dei progetti precedentemente caricati e i relativi URL per il download.

6.4.1 API WebSocket

Il protocollo WebSocket è stato utilizzato per i seguenti compiti:

- Richiedere da parte dell'utente l'URL per effettuare l'upload dello schema;
- Ricevere da parte delle funzioni le notifiche sull'avanzamento dell'elaborazione;
- Ricevere da parte delle funzioni le notifiche sugli errori generati durante l'avanzamento;
- Ricevere l'URL per poter scaricare l'archivio del progetto finale.

Il flusso delle comunicazioni sul protocollo WebSocket è mostrato in figura 6.4. Ho scelto di richiedere l'URL di upload tramite WebSocket e non tramite API HTTP RESTful in quanto ho la necessità di memorizzare l'identificatore della connessione per le successive funzioni lambda, che altrimenti non avrebbero avuto modo di comunicare con il *client*. Al contrario tutti gli altri compiti svolti con WebSocket non potevano essere implementati tramite altri protocolli. Nella fattispecie, non ho potuto utilizzare un'API HTTP RESTful in quanto avrebbe richiesto di implementare un *polling* sulle richieste dall'interfaccia utente, pratica altamente sconsigliata.

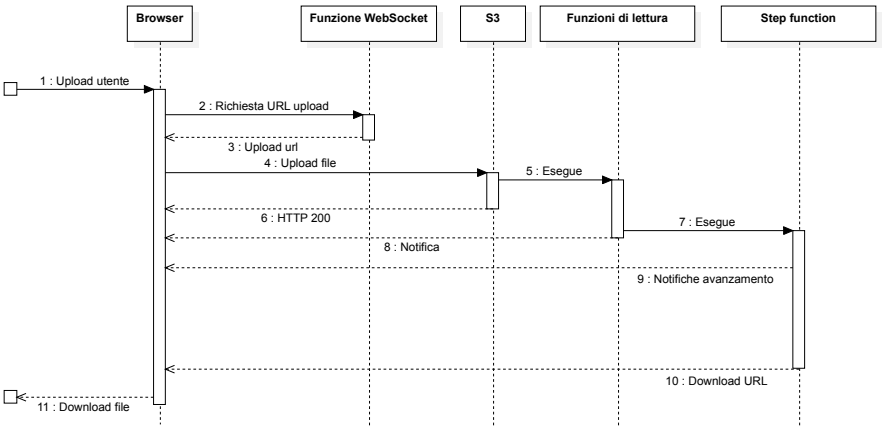


Figura 6.4: Diagramma di sequenza della comunicazione su WebSocket

Dettaglio comunicazioni Le comunicazioni WebSocket iniziano con la richiesta di un URL con il quale caricare lo schema. Il messaggio che il *client* dovrà inviare è il seguente:

```
{
    "request": "uploadUrl",
    "projectName": "project.drawio"
}
```

Successivamente la funzione Lambda che si occupa di gestire la connessione WebSocket, risponderà con il seguente messaggio:

```
{
    "projectName": "project.drawio",
    "uploadUrl": "https://develop-zero12-drawio-to-cdk..."
}
```

Come visto nel diagramma di sequenza nel paragrafo precedente, una volta che il *client* carica il file, esso si dovrà mettere solamente in ascolto delle notifiche in arrivo dalle funzioni di elaborazione. Ci sono tre tipi di notifiche, i quali seguono tutti la stessa struttura di base per semplificare l'interpretazione del messaggio da parte del *client*. il primo riguarda l'avanzamento dell'elaborazione che ha la seguente struttura:

```
{
    "Type": "notify",
    "ProjectName": "project.drawio",
    "Content": "Contenuto del messaggio"
}
```

Il secondo tipo di notifica riguarda il completamento dell'elaborazione ed ha la seguente struttura:

```
{
    "Type": "done",
    "ProjectName": "project.drawio",
    "Content": "https://develop-zero12-drawio-to-cdk..."
}
```

Il terzo tipo di notifica riguarda gli errori generati ed ha la seguente struttura:

```
{
    "Type": "done",
    "ProjectName": "project.drawio",
    "Content": "..."
}
```

Dove nel campo `Content` viene inserita una stringa formattata in JSON la quale rappresenta un array di stringhe contenenti gli errori generati.

6.4.2 API HTTP RESTful

Per completare le funzioni dell'interfaccia utente, sono state implementate delle API HTTP RESTful. Le API disponibili sono:

- `\projects-list` si occupa di ottenere la lista dei progetti elaborati dall'utente;
- `\presigned-urls` si occupa di ottenere gli URL per scaricare il diagramma e l'archivio del progetto scelto.

Lista dei progetti Effettuando una chiamata HTTP GET sulla risorsa `\projects-list` e specificando il campo `username` sulla *query string*, si ottiene una risposta con il seguente schema:

```
{
    "projects": [
        {
            "projectName": "project1",
            "version": 1
        },
        ...
    ]
}
```

URL di download Effettuando una chiamata HTTP GET sulla risorsa `\presigned-urls` e specificando i campi `username` e `projectName` sulla *query string*, si ottiene una risposta con il seguente schema:

```
{
    "project": "project1",
    "version": 1,
    "urls": {
        "schema": "https://develop-drawio-to-cdk...",
        "archive": "https://develop-drawio-to-cdk..."
    }
}
```

6.4.3 Progettazione dell'autenticazione dell'utente

L'autenticazione dell'utente viene gestita tramite il servizio AWS Cognito e schematizzata nella figura 6.5. AWS Cognito si interfaccia al servizio di autenticazione di Google per la gestione degli utenti. Utilizzare AWS Cognito si è rilevato necessario per automatizzare l'accesso alle API e non dover gestire questo aspetto interfacciandosi manualmente con Google.

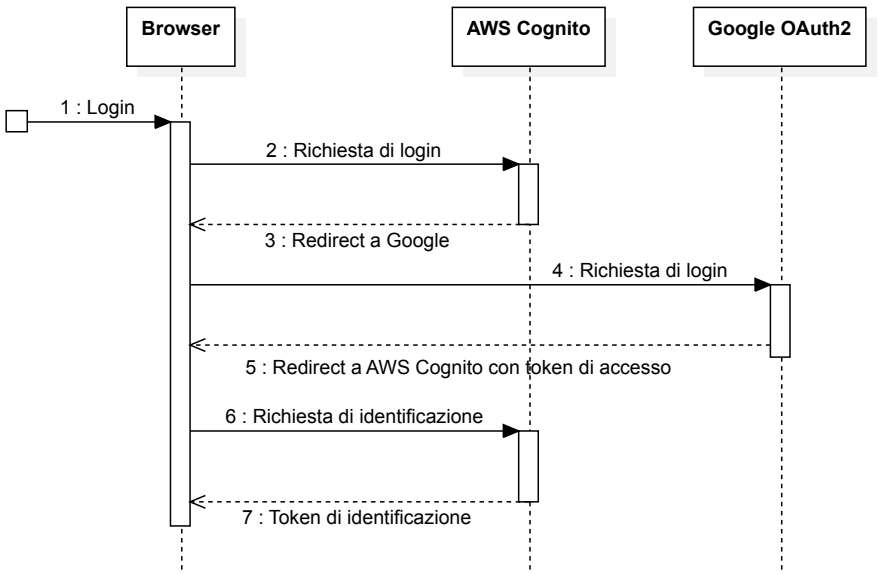


Figura 6.5: Diagramma di sequenza dell'autenticazione dell'utente.

Autenticazione delle API HTTP RESTful Per le API HTTP RESTful la verifica dell'autenticazione dell'utente è automatica e deve essere solo configurata nella definizione delle funzioni Lambda che gestiscono le varie chiamate. Quando si effettuano le chiamate alle API basterà specificare il *token* ricevuto in fase di autenticazione nell'*header* chiamato `Authorization` e la verifica avverrà in automatico. Il processo è schematizzato nella figura 6.6.

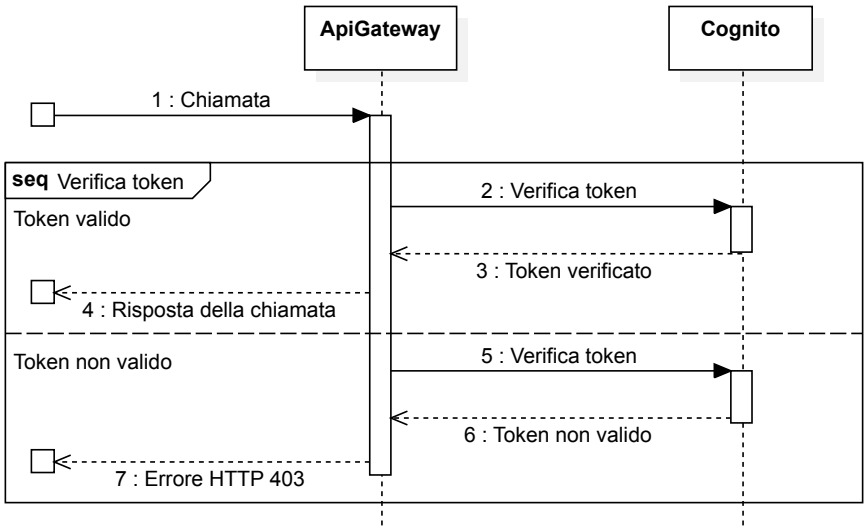


Figura 6.6: Diagramma di sequenza dell'autenticazione delle API HTTP RESTful.

Autenticazione delle API WebSocket L'autenticazione della connessione WebSocket però non funziona allo stesso modo. Questo perché l'autenticazione fornita da AWS Cognito si basa sull'ispezione dell'*header* `Authorization`. Nonostante lo *standard* WebSocket preveda la possibilità di indicare *headers* aggiuntivi per la connessione, il *client* messo a disposizione dai *browser* non lo prevede. Per questa ragione è stata aggiunta una funzione Lambda per la verifica del *token* tramite interfacciamento con AWS Cognito. Lo schema di funzionamento si trova in figura 6.7. Per comunicare il *token* al *backend*, deve essere indicato nella *query string* con il parametro `Auth`. ApiGateway si occuperà in automatico di richiamare la funzione indicata per la verifica dell'utente quando riceverà una nuova connessione WebSocket.

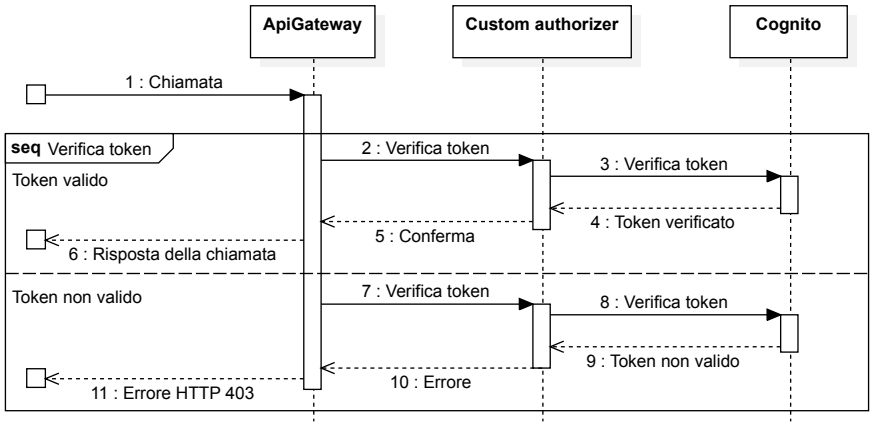


Figura 6.7: Diagramma di sequenza dell'autenticazione delle API WebSocket.

6.5 Definizione dello standard di disegno

Prima di realizzare il *software*, ho definito come le configurazioni dei servizi debbano essere inserite nel disegno. Draw.io permette di aggiungere degli attributi ai vari componenti del disegno tramite la funzione offerta chiamata "Modifica dati". Questa funzione permette di associare ad ogni componente del disegno delle informazioni in formato chiave-valore. Per le configurazioni che necessitano di dati sotto forma di liste, ho deciso di codificare le informazioni in una stringa in cui i singoli elementi sono separati da un carattere ; (*comma separated list*). Il software di conversione permette di configurare un sottoinsime dei parametri messi a disposizione dalla libreria CDK, che include tutti i parametri obbligatori. Infine Draw.io mette a disposizione diverse librerie di disegno. Per la realizzazione del progetto è stata considerata la libreria chiamata "AWS18" che, nonostante non sia la più aggiornata, contiene tutti i servizi messi a disposizione da AWS.

6.5.1 Specifica dello standard

Lo *standard* di disegno è composto da tre insiemi di regole:

- Configurazioni dei servizi;
- Posizionamento dei servizi;
- Interconnessione dei servizi.

Configurazione dei servizi Nella seguente tabella 6.1 riassumo tutte configurazioni implementate per i servizi scelti durante la pianificazione del lavoro.

Servizio	Configurazioni
Virtual Private Cloud	• `cidr`: intervallo di indirizzi IP disponibili nella rete; • `serviceNamespace`: nome di dominio locale.
VPC Subnet	• `cidr`: intervallo di indirizzi IP disponibili nella sotto-rete; • `type`: specifica se si tratta di una sotto-rete pubblica o privata; • `zone`: la zona AWS in cui la sotto-rete deve essere istanziata.
EC2	• `name`: nome per la risorsa; • `image`: l'immagine del sistema operativo da installare sull'istanza; • `cpuType`: l'architettura della CPU; • `machineClass`: il tipo di *hardware* sul quale eseguire l'istanza; • `instanceType`: la dimensione delle risorse *hardware* dedicate; • `policies`: una lista di politiche di esecuzione da associare all'istanza; • `port`: lista di porte di rete da esporre, deve avere lo stesso numero di elementi di `protocol`; • `protocol`: lista dei protocolli da associare alle porte esposte, deve avere lo stesso numero di elementi di `port`.
Lambda	• `name`: nome per la funzione; • `memory`: quantità di RAM che la funzione può utilizzare; • `ephemeralMemory`: quantità di memoria di massa che la funzione può utilizzare per i file temporanei; • `timemout`: tempo entro il quale la funzione deve terminare l'esecuzione; • `runtime`: il *runtime* di esecuzione della funzione.

Servizio	Configurazioni
S3	• `nome`: nome per la risorsa; • `publicAccess`: indica se la risorsa è accessibile pubblicamente.
DynamoDB	• `neme`: nome della risorsa.
ApiGateway	• `name`: nome della risorsa.

Tabella 6.1: Tabella riassuntiva dei possibili attributi configurabili nel disegno.

Posizionamento dei servizi Anche il posizionamento spaziale dei servizi risulta importante, non solo per il significato visivo che restituisce ma anche per il significato che assume nella configurazione. Le regole di posizionamento individuate sono riassunte nella tabella 6.2.

Servizio	Posizionamento
Virtual Private Cloud	Posizionamento libero
VPC Subnet	• Posizionata all'interno di un VPC; • Non deve sovrapporsi ad altre subnet.
EC2	Posizionato all'interno di una Subnet.
Lambda	• Posizionamento libero, senza sovrapposizione; • Posizionamento interno ad una Subnet.
S3	Posizionamento libero, all'esterno di un VPC
DynamoDB	Posizionamento libero, all'esterno di un VPC
ApiGateway	Posizionamento libero, all'esterno di un VPC

Tabella 6.2: Tabella riassuntiva delle regole di posizionamento dei blocchi di disegno.

Interconnessione tra servizi Le ultime regole importanti sono legate a come i servizi si possono collegare tra loro. Sono riassunte nella tabella 6.3. Assieme al tutor aziendale, abbiamo deciso di implementare solo le connessioni più semplici e usate. Sul disegno, le connessioni sono rappresentate da una freccia; sulla coda ci sarà il servizio utilizzatore e sulla testa quello utilizzato. Inoltre le frecce prevedono anche di indicare alcuni parametri di configurazione per la connessione tra i servizi.

Sorgente	Destinazione	Configurazioni
EC2	S3	• `grant`: Lista di permessi.
EC2	DynamoDB	• `grant`: Lista di permessi.
Lambda	S3	• `grant`: Lista di permessi.
Lambda	DynamoDB	• `grant`: Lista di permessi.
ApiGateway	Lambda	• `path`: Percorso HTTP per accedere alla risorsa; • `method`: Metodo HTTP a cui la risorsa risponde. [1]

Tabella 6.3: Tabella riassuntiva delle configurazioni delle integrazioni

[1] A riguardo di queste impostazioni, AWS supporta la possibilità di chiamare una Lambda da ApiGateway con più percorsi e metodi diversi però questa possibilità in pratica non viene mai utilizza, per questa ragione abbiamo deciso di non implementarla per mantenere più semplice il software.

Capitolo 7

Descrizione dell'applicativo

In questo capitolo descrivo il prodotto sviluppato durante lo stage.

7.1 Generazione del codice

Per generare il codice Typescript che compone l'applicazione CDK ho utilizzato la libreria messa a disposizione dal compilatore ufficiale, la quale espone una API per la creazione e manipolazione dell'albero astratto del codice. L'uso della libreria risulta necessario per ridurre al minimo il rischio di creare errori nel codice prodotto e per poter costruire in modo semplice delle funzioni che generino delle porzioni di codice predefinito o di uso diffuso.

7.1.1 Generazione della lista di importazione dei tipi

La lista di importazione dei tipi è il primo codice che si incontra in un *file* sorgente Typescript. Un *file* generato, si occupa di creare uno *stack* (vedi sezione 4.2.2), il quale potrebbe contenere diversi servizi con diverse configurazioni. Per non complicare troppo la logica di generazione del codice ho deciso di creare una lista di importazione di base per i tipi necessari allo *stack* e successivamente di creare una lista di tipi personalizzata per ogni servizio dello *stack*. Questo però non tiene conto di eventuali tipi presenti solo in alcune configurazioni particolari dei servizi stessi, ottenendo così delle liste che comprendono anche tipi non necessari all'esecuzione del codice. Per risolvere questo problema si potrebbe implementare una logica di collezionamento dei tipi necessari oppure, alternativamente un algoritmo che esplori l'albero del codice generato e collezioni i singoli tipi necessari, per generare infine la lista di importazione senza tipi inutilizzati.

7.1.2 Generazione del corpo degli stack

Gli *stack* (vedi sezione 4.2.2) sono identificati nel codice Typescript con delle classi che vengono estese per ereditarietà e nel costruttore viene inserita la logica che genera tutti i singoli servizi necessari. Nel corpo del costruttore si inserirà solo la logica per generare i servizi effettivamente presenti nel disegno. La configurazione dei servizi, dove possibile non viene scritta direttamente nel codice, ma inserita in un *file* di configurazione che verrà poi caricato dall'applicazione CDK. Ci sono però dei casi in cui la configurazione deve necessariamente essere scritta direttamente nel codice sorgente. Questo è il caso dello *stack* dedicato ad Api Gateway, il quale implementa l'integrazione con le funzioni Lambda. L'integrazione, le cui configurazioni sono consultabili nella tabella 6.3, devono necessariamente essere scritte nel codice, non potendo essere definite facilmente in un *file* di configurazione.

7.1.3 Generazione del file di configurazione

Per questo *file* la generazione risulta essere più semplice in quanto tutte le classi che lo rappresentano sono già definite nel progetto. Generando la struttura dati in base all'*input* ricevuto, sarà necessario solo richiamare le funzioni del linguaggio per esportare il *file* JSON con tutte le configurazioni.

7.1.4 Generazione delle classi di configurazione

Sviluppando l'applicazione CDK con il linguaggio Typescript è necessario fornire le definizioni dei tipi contenuti nel *file* di configurazione. Questi vengono generati rispecchiando esattamente le configurazioni presenti nel *file* descritto precedentemente. Questo permette di non avere delle definizioni di tipi superflue.

7.1.5 Generazione dell'entry point dell'applicazione

L'*entry-point* dell'applicazione CDK si occupa di istanziare le singole classi che definiscono gli *stack*, quindi anche questo *file* viene generato tenendo conto di quali *stack* sono presenti e quali no. Il meccanismo di controllo di presenza di uno *stack* è il medesimo adottato per l'esecuzione condizionale nella Step Function e schematizzato nella figura 6.3.

7.2 Architettura Serverless

Per la realizzazione del *backend* dell'applicazione ho utilizzato Serverless Framework come da *standard* aziendale. Questo mi ha permesso di concentrarmi solo sulla scrittura della *business logic* e la definizione delle risorse necessarie. La struttura del progetto

Serverless Framework, come la configurazione di tutti gli strumenti di sviluppo, è stata fornita dall'azienda sotto forma di template.

7.2.1 Definizione delle risorse

Per la definizione delle risorse, Serverless supporta sia la definizione tramite file YAML che la definizione tramite codice Typescript. Per la realizzazione del progetto ho utilizzato la definizione tramite codice Typescript in quanto, essendo tipizzato, ho potuto utilizzare le potenzialità dell'*editor* e del *linter* per semplificare la scrittura della configurazione. Le risorse seguono una nomenclatura *standard* con questa struttura:

`<configuration>-zero12-draw-io-to-cdk-<nome>`

dove:

- `configuration` può prendere uno dei seguenti valori tra: `local`, `develop`, `prod`[1];
- `nome` è il nome della risorsa specifica e deve essere univoco. [2]

7.2.1.1 Risorse di Storage

Ho utilizzato tre *bucket* S3, il primo per salvare gli schemi caricati e i progetti AWS costruiti. Il secondo è stato utilizzato per salvare i file del *template* AWS, ho scelto di separare i due *bucket* per evitare di effettuare operazioni non volute sui file del *template* che devono restare immutati. Il terzo *bucket* per servire i file del *frontend*.

Bucket per i progetti Il *bucket* chiamato `projects` contiene lo schema, l'archivio del progetto completato e in fase di elaborazione una cartella contenente tutti i file del progetto (cartella che al termine dell'archiviazione viene eliminata per risparmiare risorse). La definizione si trova nel listato 7.1. Nella definizione ho abilitato i `CORS` per permettere ai *browser* di poter caricare i file degli schemi nonostante il dominio dell'applicazione di *fronted* sia diverso da quello del *bucket*.

```
1 ProjectsBucket: {
2     Type: 'AWS::S3::Bucket',
3     Properties: {
4         BucketName: '${self:custom.${self:provider.stage}.s3
              ProjectsBucketName}',
5         CorsConfiguration: {
6             CorsRules: [
```

[1] In questo caso i tre termini sono utilizzati per indicare i diversi ambienti di esecuzione del progetto, questa differenziazione è necessaria in quanto la configurazione delle risorse non è detto che sia la stessa nei diversi ambienti. In ambito di questo progetto le configurazioni `local` e `develop` sono state utilizzate per lo sviluppo dell'applicativo, di conseguenza hanno anche alcune risorse condivise. La configurazione `prod`, essendo la configurazione di produzione, ha delle risorse dedicate e completamente isolate dalle risorse dedicate allo sviluppo.

[2] Il nome di una risorsa è univoco rispetto alle differenti configurazioni. Ad esempio una tabella di un database ha lo stesso nome ma presenta una configurazioni diverse.

```
                {
                    AllowedHeaders: [
                            '*'
                    ],
                    AllowedMethods: [
                        'GET',
                        'PUT',
                        'DELETE',
                        'HEAD',
                        'POST'
                    ],
                    AllowedOrigins: [
                        '*'
                    ]
                }
            ]
        }
    }
}
```

Listato 7.1: Codice di configurazone del Bucket S3 per i progetti.

Bucket per il template Il *bucket* chiamato *template* contiene un'app CDK vuota con annessi alcuni *file custom* di configurazione e del codice statico per la lettura dei *file* di configurazione. La definizione si trova nel listato 7.2. In questo non ho dovuto abilitare i `CORS` perché questo *bucket* viene acceduto solo dal *backend* e non deve interagire con altri sistemi esterni.

```
TemplateBucket: {
    Type: 'AWS::S3::Bucket',
    Properties: {
        BucketName: '${self:custom.${self:provider.stage}.s3
            TemplateBucketName}'
    }
}
```

Listato 7.2: Definizione del bucket del template del progetto

Bucket per il frontend Ho utilizzato un *bucket* dedicato per distribuire i *file* statici dell'interfaccia dell'applicativo. I file in questo *bucket* vengono acceduti tramite CloudFront che si occupa di rendere disponibile all'*endpoint* `https://drawiocdk.zero12.co` l'applicazione. In questo caso non è presente una configurazione per il *Serverless Framework* in quanto non viene utilizzato dal *backend*. Il *bucket* viene popolato dalla CI/CD del *frontend*.

7.2.1.2 Risorse di Database

Come *database* ho utilizzato DynamoDB che permette di creare delle tabelle nelle quali è necessario solo specificare la chiave primaria. Tutti gli altri attributi sono dinamici e

possono essere aggiunti e tolti a bisogno. Nella realizzazione del progetto ho utilizzato due tabelle.

Tabella di associazione progetto-connessione Ho utilizzato una tabella per associare ad ogni progetto in elaborazione, il proprio identificatore di connessione fornito da ApiGateway. Ho trovato necessario creare un'associazione tra la connessione WebSocket e il nome del progetto per tenere traccia del client al quale mandare le notifiche sull'avanzamento dell'elaborazione. La definizione si trova nel listato 7.3. In questa tabella è utile notare come ho inserito un secondo indice. L'indice secondario mi è tornato utile per cercare dal nome del progetto, l'identificatore di connessione sul quale mandare i messaggi. Essendo che l'indice secondario non lavora sulla chiave primaria della tabella, ho specificato gli attributi utilizzati dall'indice nell'apposito attributo.

```
ConnectionIdTable: {
    Type: 'AWS::DynamoDB::Table',
    Properties:{
        TableName: '${self:custom.${self:provider.stage}.
            DynamoDBConnectionIdTable}',
        AttributeDefinitions: [
            {
                AttributeName: 'connectionId',
                AttributeType: 'S'
            },
            {
                AttributeName: 's3ObjectKey',
                AttributeType: 'S'
            }
        ],
        KeySchema: [
            {
                AttributeName: 'connectionId',
                KeyType: 'HASH'
            }
        ],
        GlobalSecondaryIndexes: [
            {
                IndexName: 'S3ObjectIndex',
                KeySchema: [
                    {
                    AttributeName: 's3ObjectKey',
                    KeyType: 'HASH'
                    },
                ],
                Projection: {
                    ProjectionType: 'ALL'
                },
                ProvisionedThroughput: {
                    ReadCapacityUnits: 1,
                    WriteCapacityUnits: 1
```

```
                }
            }
        ],
        ProvisionedThroughput: {
            ReadCapacityUnits: 1,
            WriteCapacityUnits: 1
        }
    }
}
```

Listato 7.3: Codice di configurazione della tabella di associazione progetto-connessione

Tabella di associazione utente-progetto Nell'interfaccia utente ho pensato di inserire la lista di progetti caricati dall'utente, questo per permettere all'utente di accedere ai vecchi progetti. Per questo motivo ho introdotto una tabella per associare ad ogni utente registrato dal servizio AWS Cognito, i progetti da esso elaborati. La sua definizione si trova nel listato 7.4. Da notare che, anche in questo caso, ho inserito un indice secondario per poter cercare tutti i progetti di un utente.

```
ProjectsListTable: {
    Type: 'AWS::DynamoDB::Table',
    Properties:{
        TableName: '${self:custom.${self:provider.stage}.
            DynamoDBProjectsListTable}',
        AttributeDefinitions: [
            {
                AttributeName: 'username',
                AttributeType: 'S'
            },
            {
                AttributeName: 'projectName',
                AttributeType: 'S'
            }
        ],
        KeySchema: [
            {
                AttributeName: 'username',
                KeyType: 'HASH'
            },
            {
                AttributeName: 'projectName',
                KeyType: 'RANGE'
            }
        ],
        GlobalSecondaryIndexes: [
            {
                IndexName: 'UsernameIndex',
                KeySchema: [
                    {
                        AttributeName: 'username',
                        KeyType: 'HASH'
                    }
```

```
                ],
                Projection: {
                    ProjectionType: 'ALL',
                },
                ProvisionedThroughput: {
                    ReadCapacityUnits: 1,
                    WriteCapacityUnits: 1
                }
            }
        ],
        ProvisionedThroughput: {
            ReadCapacityUnits: 1,
            WriteCapacityUnits: 1
        }
    }
}
```

Listato 7.4: Codice di configurazione della tabella di associazione utente-progetto

7.2.1.3 Servizio di autenticazione

Ho utilizzato una *user pool* di AWS Cognito per gestire l'autenticazione degli utenti sull'interfaccia *web* e per l'accesso alle API. Per la definizione di questa risorsa ho utilizzato una configurazione manuale per poter abilitare durante il processo di creazione l'autenticazione tramite *account* Google. Nel sorgente del progetto tale risorsa è identificata tramite l'ARN, un codice univoco generato da AWS per l'identificazione della risorsa.

7.2.2 Definizione e descrizione delle funzioni Lambda

Avendo sviluppato tutto seguendo l'architettura *serverless* ho utilizzato le funzioni Lambda per implementare la *business logic*. Per rispettare la regola del *single responsibility principle* ho creato una Lambda per ogni compito computazionale. La definizione mi ha richiesto di specificare un nome e di indicare dove si trova il codice da eseguire. Di particolare interesse la definizione degli eventi che possono scatenare l'esecuzione della Lambda: posso specificare una lista di eventi, anche eterogenei tra di loro, che scatenano l'esecuzione della funzione. Nel mio caso sono solo stati intercettati gli eventi emessi da S3, da WebSocket e da HTTP. D'altro canto le funzioni eseguite dalla Step Function non necessitano di essere avviate da un evento, in quanto è l'orchestratore stesso ad occuparsene. Nel listato 7.5 un esempio di definizione di una funzione per la Step Function.

```
fileConverter: {
    handler: `${handlerPath(__dirname)}/handlers/fileConverter.
        main`,
    name: '${self:provider.stage}-${self:service}-fileConverter',
    timeout: 25,
```

```
    memorySize: 512
}
```

Listato 7.5: Definizione di una funzione Lambda per l'esecuzione nella Step Function

7.2.2.1 Lettura del file ed estrazione delle informazioni

La prima Lambda che ho realizzato si occupa di leggere il contenuto del file e di estrarne le informazioni inserite dall'utente. Questa funzione viene lanciata direttamente dopo il caricamento di un file su S3. Questa Lambda può generare degli errori che ho comunicato all'utente. Per la lettura del file di Draw.io ho utilizzato la libreria `xml2js` che si occupa di convertire un file XML in un oggetto Javascript. Utilizzando però Typescript ho dovuto anche definire tutte le classi che rappresentano tutti i tag XML presenti nel disegno. Questo come descriverò in seguito ha delle implicazioni per una possibile estensione del software. La definizione della funzione è nel listato 7.6

```
fileReader: {
    handler: `${handlerPath(__dirname)}/handlers/fileReader.
        handler`,
    name: '${self:provider.stage}-${self:service}-fileReader',
    events: [
        {
            s3: {
                bucket: '${self:custom.${self:provider.stage}.s3
                    ProjectsBucketName}',
                existing: true,
                event: 's3:ObjectCreated:*',
                rules: [{
                    suffix: '.drawio'
                }]
            }
        }
    ],
    timeout: 25,
    memorySize: 512
}
```

Listato 7.6: Definizione della Lambda di lettura del file.

7.2.2.2 Validazione e conversione delle informazioni

La seconda Lambda che ho implementato si occupa di validare i dati provenienti dal disegno e di convertirli in una struttura dati che rispecchi l'infrastruttura rappresentata. Questa Lambda può generare degli errori che ho ritornato poi all'utente. La definizione di questa funzione è analoga a quella definita nel listato 7.5.

7.2.2.3 Creazione del progetto

La terza Lambda che ho implementato si occupa di copiare il template dal *bucket* dedicato a quello che contiene il progetto in fase di elaborazione. La definizione di questa funzione è equivalente a quella definita nel listato 7.5.

7.2.2.4 Generazione del codice

Questo è l'insieme di Lambda che si occupa di generare le diverse parti del codice. Avendo utilizzando un sistema di nomi univoci generati seguendo delle regole fisse, ho potuto realizzare una Lambda separata per ogni file di codice da generare. Durante lo sviluppo di queste Lambda ho incontrato diversi problemi per la scrittura della parte dedicata alla generazione del codice Typescript dell'applicazione CDK. Pur avendo scelto la libreria del compilatore ufficiale, la documentazione è molto scarsa e copre solo alcuni esempi banali di generazione del codice. Per questa ragione ho esplorato le funzionalità della libreria alla cieca, deducendone il funzionamento basandomi sulla conoscenza della grammatica del linguaggio. Inoltre ho trovato utile creare un insieme di funzioni che incapsulano le funzioni esposte dalla libreria per facilitare la scrittura successiva del codice. La definizione di questa funzione è equivalente a quella definita nel listato 7.5.

7.2.2.5 Archiviazione del progetto

Infine la Lambda che chiude il processo di elaborazione, si occupa di archiviare in formato ZIP tutti i file generati. Ad archiviazione completata, pulisce il *bucket* dei progetti dai file del progetto e manda una notifica di completamento con il *link* per scaricare l'archivio. La definizione di questa funzione è equivalente alla definizione nel listato 7.5.

7.2.2.6 Comunicazioni WebSocket

Le comunicazioni attraverso il protocollo WebSocket sono gestite da un'unica funzione Lambda. In questo caso ho scelto di violare il *single responsibility principle* unificando la gestione dei tre diversi eventi generati dal protocollo WebSocket in un'unica funzione Lambda (vedi Appendice A per approfondire). La definizione di questa funzione è nel listato 7.7.

```
1 websocket: {
2     handler: `${handlerPath(__dirname)}/handlers/websocket.main`,
3     name: '${self:provider.stage}-${self:service}-websocket',
4     events: [
5         {
6             websocket: {
7             route: '$connect',
8             authorizer: {
```

```
                    name: 'authorizer',
                    identitySource: ['route.request.querystring.
                        Auth']
                }
            }
        },
        {
            websocket: {
                route: '$default'
            }
        },
        {
            websocket: {
                route: '$disconnect'
            }
        }
    ],
    timeout: 25,
    memorySize: 128
}
```

Listato 7.7: Definizione di una Lambda con eventi da WebSocket.

7.2.2.7 Lista progetti

Questa Lambda risponde ad una chiamata HTTP GET. Si occupa di ottenere dal database dei progetti, tutti quelli associati all'utente che sta eseguendo la richiesta. La richiesta ritornerà un errore se non si specifica un utente. Inoltre questa chiamata è protetta dall'autenticazione gestita da ApiGateway; è necessario indicare l'*header Authentication* con l'*ID Token* ottenuto con l'autenticazione tramite AWS Cognito. La definizione di questa funzione è nel listato 7.8.

```
projectsList: {
    handler: `${handlerPath(__dirname)}/handlers/projectsList.
        main`,
    name: '${self:provider.stage}-${self:service}-projects-list',
    events: [
        {
            http: {
                method: 'GET',
                path: '/projects-list',
                cors: true,
                authorizer: {
                    arn: '${self:custom.${self:provider.stage}.
                        cognitoPoolArn}'
                }
            }
        }
    ],
    timeout: 25,
    memorySize: 512
}
```

Listato 7.8: Definizione di una Lambda che risponde agli eventi HTTP.

7.2.2.8 Ottenimento dei Presigned URL

Questa Lambda risponde ad una chiamata API GET. Si occupa di generare dei *presigned url* (vedi Appendice B per approfondire) per scaricare dal *bucket* dei progetti l'archivio ZIP dell'applicazione CDK e il disegno Draw.io. Questa Lambda richiede l'utente e nome del progetto per poter ritornare correttamente gli URL. In caso contrario restituirà errore. Restituirà errore anche nel caso il progetto indicato non si tra quelli in possesso dell'utente. Inoltre questa chiamata è protetta dall'autenticazione gestita da Api Gateway; è necessario indicare l'*header Authentication* con l'*ID Token* ottenuto con l'autenticazione tramite AWS Cognito. La definizione di questa funzione è analoga alla definizione nel listato 7.8.

7.2.2.9 Funzione di autenticazione

Tutte le API pubbliche sono state protette da autenticazione tramite AWS Cognito. La connessione WebSocket (gestita nell'apposita funzione descritta nella sezione 7.2.2.6), essendo anch'essa una API, non è da meno. In questo caso, non potendo specificare degli *Headers* personalizzati per la connessione dal *browser*, ho inserito il *token* di autenticazione direttamente sull'URL della connessione e tramite questa funzione Lambda, mi sono interfacciato con il servizio AWS Cognito controllare che il *token* sia valido. Questa funzione ha la seguente definizione:

```
1 authorizer: {
2     handler: `${handlerPath(__dirname)}/handlers/authorizer.
          handler`,
3     name: '${self:provider.stage}-${self:service}-authorizer',
4     environment: {
5         COGNITO_POOL_ID: '${self:custom.${self:provider.stage}.
              cognitoPoolId}'
6     },
7     timeout: 25,
8     memorySize: 128
9 },
```

7.2.2.10 Implementazione della Step Function

Il *framework* Serverless supporta la definizione delle Step Function ma solo tramite *file* di configurazione YAML. Nel mio progetto però sto utilizzando Typescript per la definizione della configurazione del *framework* Serverless, quindi per poter scrivere la definizione tramite Typescript ho integrato la definizione delle classi relative nel progetto. Questo mi ha permesso di sfruttare comunque tutte le potenzialità di Visual

Studio Code nella verifica e suggerimento del codice, che altrimenti non avrei avuto, aiutandomi nella stesura di tutta la definizione. La definizione della Step Function, consiste nel definire i singoli stati. Gli stati però non sono solo delle funzioni (vedi listato 7.9) ma possono anche specificare il flusso dei esecuzione della macchina stessa, come un'esecuzione parallela (vedi listato 7.10) oppure un'esecuzione condizionale (vedi listato 7.11).

```
'DrawIoToInfrastructure': {
    Type: 'Task',
    Resource: {
        'Fn::GetAtt': ['fileConverter', 'Arn']
    },
    Next: 'CreateProject'
}
```

Listato 7.9: Definizione di uno stato che esegue una funzione

```
'CreateProjectFiles': {
    Type: 'Parallel',
    Next: 'ArchiveProject',
    Branches: [
        {
            StartAt: 'CreateCdkJson',
            States: {
                'CreateCdkJson': {
                    Type: 'Task',
                    Resource: {
                        'Fn::GetAtt': ['createConfig', 'Arn']
                    },
                    End: true
                }
            }
        },
        ...
    ]
}
```

Listato 7.10: Definizione di uno stato di esecuzione parallela

```
CreateComputeStackChoice: {
    Type:'Choice',
    Default: 'ComputeStackPass',
    Choices: [
        {
            Variable: '$.infrastructure.computeStack',
            BooleanEquals: true,
            Next: 'CreateComputeStack',
        }
    ]
},
'ComputeStackPass': {
    Type:'Pass',
    End: true
```

```
15 },
16 'CreateComputeStack': {
17     Type: 'Task',
18     Resource:{
19         'Fn::GetAtt': ['computeCreator', 'Arn']
20     },
21     End: true
22 }
```

Listato 7.11: Definizione di uno stato di esecuzione condizionale

7.3 Interfaccia web

Ho sviluppato l'interfaccia *web* come *single page application*, utilizzando il *framework* React. Per mantenere il progetto semplice ho deciso di non includere librerie di terze parti che implementano *routing* o componenti grafiche. Ho utilizzato React solo per agevolare il rendering della pagina. Ho preso questa decisione assieme al tutor aziendale in quanto l'applicazione è molto semplice.

7.3.1 Login

Ho implementato il *login* con Google utilizzando la libreria Amplify per interfacciare l'applicazione con il servizio AWS Cognito nel *backend*. L'implementazione è stata fatta sul componente principale dell'applicazione, in questo modo ho assicurato che tutte le altre pagine dell'applicazione possano essere visualizzate solo se l'utente è autenticato.

7.3.2 Upload del file dello schema

La pagina di *upload* del *file* è composta da un *form* che prevede la selezione del *file* e successivamente l'inserimento del nome del progetto. Il *file* può essere selezionato anche tramite *drag and drop*, mentre il nome del progetto è di *default* il nome del *file* selezionato.

7.3.3 Schermata di avanzamento dell'elaborazione

Una volta caricato il *file*, viene mostrata la pagina di avanzamento dell'elaborazione. Per disaccoppiare il *frontend* dal *backend* ho deciso di non mostrare una percentuale di avanzamento. Al contrario mostro all'utente una lista di notifiche che compaiono durante le varie fasi dell'elaborazione.

7.3.4 Schermata di download del progetto

Una volta completata l'elaborazione, mostro direttamente una pagina dalla quale posso scaricare il progetto e tornare alla pagina di *upload*.

7.3.5 Schermata degli errori

Se l'elaborazione genera degli errori, questi vengono mostrati immediatamente all'utente, nascondendo la pagina di avanzamento dell'elaborazione. Gli errori vengono lanciati indipendentemente dalle varie Lambda, quindi anche questa schermata mostra gli errori a mano a mano che vengono ricevuti.

7.3.6 Lista dei progetti

Infine la lista dei progetti già generati. Questa pagina si seleziona dal menù di navigazione accessibile premendo sulla foto profilo. In questa pagina è visibile una lista di progetti generati in passato nella quale, premendo su uno di essi, compare un dialogo in sovra impressione, il quale permette di scaricare sia il disegno che l'archivio con i file di progetto.

7.4 CLI

Avendo avanzato del tempo alla fine dello sviluppo ho proposto la creazione di un CLI per interfacciarsi con l'applicazione. La proposta è stata accettata positivamente dall'azienda. La CLI è stata sviluppata anch'essa con Typescript. La CLI è pensata per sostituire integralmente l'interfaccia *web*, quindi propone le stesse funzionalità. Nella seguente tabella riassumo brevemente i comandi disponibili e le eventuali opzioni disponibili:

7.4.1 Libreria per la gestione degli argomenti

Per gestire gli argomenti passati con la *console* ho utilizzato una libreria chiamata `yargs`. Tale libreria permette di costruire un *parser* che legge e converte gli argomenti *console* automaticamente, specificandone anche il significato. Così facendo la libreria crea anche il comando `help` automaticamente per istruire l'utente all'utilizzo della CLI.

7.4.2 Funzionamento del login

Il *login* è fondamentale per ottenere un *token* di accesso alle API. Utilizzando AWS Cognito, cioè un *provider* OAuth2, mi sono interfacciato tramite gli *endpoint* esposti. Per eseguire il *login* faccio aprire il *browser* di sistema sull'*endpoint* di AWS Cognito `/oauth2/authorizer` che rimanda alla pagina di *login* con Google. Nel caso il *browser*

Comando	Descrizione	Opzioni disponibili
`login`	Esegue il *login* con Google per accedere alle API	Nessuna
`convert`	Converte il disegno di Draw.io nel *template* CDK	`-pn` oppure `--project-name` permette di specificare un nome per il progetto se differisce dal nome del file dello schema Draw.io
`list`	Visualizza la lista di tutti i progetti convertiti	`-q` oppure `--query` permette di indicare una stringa con cui filtrare i nomi di progetto
`download`	Permette di scaricare un progetto, presenta due modalità, una interattiva nella quale si mostra la lista di progetti per effettuare la scelta, la seconda permette di specificare il nome del progetto da scaricare. Di default scarica l'archivio del progetto CDK	• `-a` oppure `--archive` permette di indicare che si vuole che si vuole scaricare il progetto CDK • `-d` oppure `--draw` permette di indicare che si vuole scaricare lo schema Draw.io

Tabella 7.2: Tabella riassuntiva dei comandi ed opzioni forniti dalla CLI.

non si apra in automatico, l'utente può comunque aprire la pagina di *login* premendo sul *link* che compare sulla *console*. Nel frattempo la CLI apre un *server* HTTP in locale che attende la risposta da AWS Cognito. Appena l'utente inserisce le credenziali o sceglie il profilo, vengono generati i *token* di accesso e la pagina esegue un reindirizzamento sull'URL specificato. L'URL specificato in questo caso è quello esposto dalla CLI. A questo punto la CLI si salva le informazioni del *login* in un file alla *path* `~/.zero12/draw-io-to-cdk/userinfo.json`. I *token* di accesso infatti sono validi per 1 ora, così facendo, utilizzando più volte il comando non ci si dovrà autenticare più volte.

7.4.3 Le altre funzionalità

Per l'implementazione delle altre funzionalità sono state utilizzate le stesse API utilizzate dell'interfaccia *web*, quindi il funzionamento è lo stesso.

7.4.4 Installazione del comando

Utilizzando Node.js e NPM per la gestione dei pacchetti, la compilazione e l'installazione della CLI in locale risulta molto semplice. La compilazione infatti è gestita tramite il comando `npm build`, che si occupa di compilare tutti i file di progetto. Con il comando `npm i -g .` si installa la CLI nel sistema a livello globale. Il file `packet.json` specifica

quale file è l'*entry point* per l'esecuzione del *software*. L'unico accorgimento necessario per far funzionare correttamente la CLI è indicare nell'*entry point*, sulla prima riga del file, qual è il file eseguibile che il sistema operativo deve usare per eseguire lo script. Nel mio caso è Node.js il quale si indica con la riga `#!/usr/bin/env node`.

Capitolo 8

Uso del prodotto

In questo capitolo vengono discusse alcune considerazioni finali sull'uso del prodotto sviluppato.

8.1 Considerazioni sull'uso pratico

Questo progetto, come visto nella sezione 1.2, nasce per rispondere ad un'esigenza reale all'interno dell'azienda. Lo sviluppo si è concentrato principalmente per rendere il prodotto utilizzabile in un contesto reale. In riferimento a quanto detto nell'introduzione, il prodotto porta il seguente valore aggiunto ad uno schema architetturale:

- Possibilità di avere sempre disegno e *template* CDK sincronizzati;
- Convenzioni e *standard* sempre rispettati sul codice generato;
- Velocità di implementazione del *template*;
- Possibilità di personalizzare il *template* di base;
- Non sono necessarie competenze di programmazione per applicare l'approccio *Infrastructure as Code*.

Dal'altro canto però, l'uso di questo software porta anche delle controindicazioni:

- Schemi infrastrutturali diventano più complessi;
- Difficile da integrare in una *pipeline* CI/CD;
- Non è possibile definire più *set* di configurazioni sullo stesso disegno.

8.2 Considerazioni sullo sviluppo futuro del prodotto

Il prodotto sviluppato include pochi servizi e poche integrazioni tra essi. Per questa ragione per poter utilizzare il prodotto in un contesto reale sarà prima necessario un ulteriore sviluppo ed espansione. I punti individuati su cui è necessario lavorare sono:

- Possibile cambio della libreria di generazione del codice in favore di una meglio documentata;
- Uso di strutture dati che agevolino l'esplorazione e la ricerca dei blocchi del disegno;
- Integrazione di altri servizi AWS;
- Gestione migliore delle istruzioni di `import` per evitare importazioni di tipi non utilizzati nel codice;
- Implementazione di una formattazione del codice adeguata agli standard aziendali;
- Applicazione del paradigma *convention over configuration*, identificando un set di impostazioni di default per la configurazione, in modo da far specificare all'utente solo le configurazioni necessarie.

Capitolo 9

Conclusioni

In questo capitolo sono presentate le conclusioni riguardo al progetto.

9.1 Consuntivo finale

Nella tabella 9.1 sono riportate le attività svolte durante le 320 ore di stage, correlate da un commento che spiega la discrepanza con la pianificazione.

Periodo	Durata (ore)	Attività
1	28	Studio delle tecnologie AWS e organizzazione del *software* a micro-servizi
2	32	Studio e creazione di architetture *cloud* tramite Draw.io e studio del formato del file di esportazione
3	100	Progettazione e sviluppo dell'algoritmo che dato il disegno in *input* genera il template CDK e sviluppo API
4	40	Creazione dell'infrastruttura e *deploy* tramite servizi di CI/CD
5	40	Creazione dell'interfaccia *web*
6	40	Scrittura documentazione, sul codice e su piattaforma aziendale. *Test* e validazione del prodotto.
7	40	Scrittura interfaccia CLI

Tabella 9.1: Consuntivo finale.

Il consuntivo finale ha delle sostanziali differenze con la pianificazione iniziale del lavoro. In particolare, per ogni periodo:

1. La formazione è stata particolarmente breve in quanto avevo già dimestichezza con il linguaggio di programmazione scelto e con le metodologie di lavoro adottate dall'azienda;
2. Lo studio del formato di esportazione del disegno di Draw.io sembrava inizialmente più complicato, ma così non è stato; utilizzando soprattutto degli strumenti automatici per l'estrazione delle strutture dati necessarie dal disegno, il lavoro è stato svolto agevolmente;
3. La creazione dell'algoritmo di conversione ha richiesto un lavoro maggiore rispetto alle previsioni, in quanto ho dovuto lavorare con librerie poco documentate che hanno richiesto uno sforzo maggiore;
4. La creazione dell'infrastruttura è stata più veloce del previsto nonostante ha richiesto di integrare tecnologie non previste;
5. Ho potuto curare l'interfaccia *web* grazie all'avanzamento veloce nei periodi precedenti;
6. I *test* sono stati svolti nei disegni di infrastruttura prodotti nel periodo 2; sono stati svolti come test di integrazione del prodotto;
7. Su mia proposta ho sviluppato anche un'interfaccia CLI, proposta accettata dall'azienda visto l'anticipo sul completamento delle attività previste.

9.2 Raggiungimento degli obiettivi

Gli obiettivi sono stati tutti raggiunti. Il tempo preventivato era adeguato per tutte le attività ed ha tenuto conto delle possibili difficoltà che potevano emergere durante il lavoro. L'unica attività che ha richiesto più di quanto preventivato è stata la creazione dell'algoritmo di conversione viste le difficoltà incontrate con le librerie. Oltre agli obiettivi posti dall'azienda per il completamento del progetto, è stato aggiunta anche l'interfaccia a riga di comando da me proposta.

9.3 Conoscenze acquisite

Durante le otto settimane di stage ho potuto apprendere moltissimo riguardo al mondo *cloud*, vedendo come molti dei concetti appresi durante l'università si applicano tutti i giorni nel mondo reale. Tra le conoscenze di maggior rilievo cito:

- Generazione di codice;

- Progettazione e implementazione di un'applicazione *serverless*;
- Progettazione e implementazione di un *frontend* che rispetti la sicurezza degli utenti;
- Progettazione e implementazione di interfacce a riga di comando.

Tra le conoscenze acquisite non ci sono solo *hard-skills* ma anche delle *soft-skills*. Tra le quali la comunicazione ritengo sia la più importante. Ho imparato a comunicare in modo chiaro ed efficace le mie idee e difficoltà per agevolare il *team*.

9.4 Valutazione personale

Questo stage mi ha permesso di consolidare le conoscenze acquisite durante il percorso di studi e di ampliarle, potendo vedere e capire come vengono applicate nel mondo reale. Ho avuto la possibilità di sperimentare e dare sfogo alla mia creatività nel trovare una soluzione ad un problema poco comune. Ritengo sia stata un'esperienza molto positiva e che mi ha fatto crescere molto a livello professionale.

Acronimi e abbreviazioni

API Application Program Interface. 3, 4, 11, 12, 14, 18, 20, 21, 26, 32, 36, 39, 40, 44, 48, 49, 53

ARN Amazon Resource Names. 32

AWS Amazon Web Services. 2–4, 7–9, 11–14, 20, 22, 23, 28, 31, 32, 35, 36, 38–40, 43, 44, 48, 52, 53

CDK Cloud Development Kit. 2, 4, 8–10, 22, 26, 27, 29, 34, 36, 40, 42, 44

CI/CD Continuous Integration/Continuous Delivery. 5, 14, 29, 42, 44

CLI Command Line Interface. 9, 10, 39–41, 44, 45

HTTP Hyper Text Transfer Protocol. 3, 12, 14, 18, 20, 21, 32, 35, 40, 48, 53

IaC Infrastructure as Code. 48

JSON Javascript Object Notation. 10, 19, 27

OOP Object Oriented Programming. 11

RESTful Representational State Transfer. 3, 12, 14, 18, 20, 21, 49

URL Uniform Resource Locator. 18–20, 36, 40, 49, 53

VPC Virtual Private Cloud. 24

XML Extensible Markup Language. 33

YAML YAML Ain't Markup Language. 11, 28, 36

Glossario

Amazon Resource Names In ambito AWS con il termine *Amazon Resource Names* (ing. Nome Risorsa Amazon) si intende un nome univoco che identifica una specifica risorsa istanziata nell'infrastruttura AWS. L'ARN codifica al suo interno delle informazioni tra cui il *data center*, l'*account*, la regione e il tipo di risorsa.. 47

API in informatica con il termine *Application Programming Interface API* (ing. interfaccia di programmazione di un'applicazione) si indica ogni insieme di procedure disponibili al programmatore, di solito raggruppate a formare un set di strumenti specifici per l'espletamento di un determinato compito all'interno di un certo programma. La finalità è ottenere un'astrazione, di solito tra l'hardware e il programmatore o tra software a basso e quello ad alto livello semplificando così il lavoro di programmazione. 47

Continuous Integration/Continuous Delivery In informatica con il termine *Continuous Integration/Continuous Delivery* (ing. Integrazione continua/ rilascio continuo) si intende una pratica di sviluppo per cui tutte le modifiche apportate durante lo sviluppo di un *software* vengono provate, integrate e rilasciate appena pronte, senza aspettare anche tutte le modifiche previste per una *milestone* siano anch'esse pronte.. 47

Command Line Interface In informatica con il termine *Command Line Interface* (ing. Interfaccia a linea di comando) si indica un'applicazione che per interfacciarsi con l'utente utilizza un'interfaccia completamente testuale, spesso visualizzata tramite un terminale.. 47

HTTP In informatica con il terminate *Hyper Text Transfer Protocol* (ing. protocollo di trasferimento di ipertesto) si indica un protocollo che si occupa di gestire la trasmissione tra client e server di pagine e informazioni di tipo ipertestuale. 47

IaC In informatica con il termine *Infrastructure as code* (ing. infrastruttura come codice) si indica un paradigma di gestione dell'infrastruttura basato sulla scrittura

di code sorgente in un linguaggio di programmazione, oppure sulla scrittura di un file di configurazione che segue delle regole previste dal fornitore di servizi cloud. 3, 7, 8, 42, 47

Javascript Object Notation In informatica con il termine *Javascript Object Notation* (ing. Notazione degli oggetti Javascript) si indica la rappresentazione utilizzata nel linguaggio di programmazione Javascript per rappresentare su una stringa le strutture dati.. 47

Linter In informatica con il termine *linter* o *lint* si indica un *software* di analisi statica di codice atto a trovare errori di programmazione, errori stilistici, costrutti sospetti, o porzioni di codice malevole.. 28

Micro-servizi In informatica con il termine *micro-servizi* si intende un approccio di progettazione e sviluppo di un'applicazione nel quale l'applicazione viene divisa in piccole parti composte da servizi indipendenti che comunicano tra di loro tramite API ben definite. Le applicazioni secondo queste regole sono tendenzialmente più facili da scalare rispetto alle controparti monolitiche.. 4, 44

REST In informatica con il termine *Representiational State Transfer REST* si indica un paradigma di sviluppo di API basato sul protocollo HTTP, che si pone come obiettivo lo scambio di informazioni tra *client* ed *server* in modo sicuro. Una API RESTful prevede che tutte le chiamate siano identificate univocamente, che siano stateless e che il server invii tutti i metadati necessari al client per interpretare e completare l'operazione richiesta.. 47

Rollback In informatica con il termine *rollback* (ing. tornare indietro) si intende un'operazione che prevede di ripristinare ad uno stato o versione precedente un qualche sistema.. 7

URL In informatica con il termine *Uniform Resource Locator* (ing. Localizzatore uniforme di risorse) si indica una stringa di testo strutturata secondo lo standard RTC 3986 la quale identifica una univocamente una risorsa messa a disposizione da un computer su una rete specificando protocollo, destinazione, percorso della risorsa e altre informazioni.. 47

WebSocket In informatica con il termine *WebSocket* si indica il protocollo per lo scambio di messaggi asincroni tra un *server* e un *client*, condivide con HTTP la fase di *handshacke* e la porta 80/TCP usata per la connessione. Il protocollo *WebSocket* e stato standardizzato dal W3C come RFC 6455.. 12, 14, 16, 18, 19, 22, 30, 32, 34, 36

Extensible Markup Language In informatica con il termine *Extensible Markup Language* (ing. Linguaggio di marcatura estensibile) si indica un metalinguaggio per la definizione di linguaggi di marchiatura, ovvero un linguaggio basato su un meccanismo sintattico che consente di definire e controllare il significato degli elementi contenuti in un documento o testo.. 47

YAML Ain't Markup Language In informatica con il termine *YAML Ain't Markup Language* (ing. YAML non è un linguaggio di marcatura) si indica un formato per la serializzazione dei dati utilizzabile da esseri umani. Il nome deriva dalla volontà dei creatori di indicare che lo scopo del linguaggio fosse la memorizzazione dei dati.. 47

ZIP In informatica con il termine *ZIP* si intende un formato di file che archivia e comprime dati di vario genere. La compressione che applica è di tipo *lossless*, permettendo dunque di preservare il file originale. Supporta diversi algoritmi di compressione, il più utilizzato è *DEFLATE*.. 34, 36

Bibliografia

Amazon Web Services. *What is Amazon API Gateway?* URL: https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html. Ultima verifica: 11/09/2023.

— *What is Amazon Cognito?* URL: https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html. Ultima verifica: 11/09/2023.

— *What is Amazon DynamoDB?* URL: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html. Ultima verifica: 11/09/2023.

— *What is Amazon S3?* URL: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html. Ultima verifica: 11/09/2023.

— *What is AWS Lambda?* URL: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html. Ultima verifica: 11/09/2023.

— *What is AWS Step Functions?* URL: https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html. Ultima verifica: 11/09/2023.

Dindinelli, Alessandro. *INFRASTRUCTURE AS CODE (IAC): COS'È?* URL: https://www.zero12.it/infrastructure-as-code-iac/?_gl=1*ld9qyg*_ga*NTQ1MjY4MTAyLjE2OTQ0MjQ0OTE.*_up*MQ..*_ga_EMNK7KNX8F*MTY5NDQyNDQ5MC4xLjEuMTY5NDQyNDQ5My4wLjAuMA..*_ga_F6C07J2PGB*MTY5NDQyNDQ5MC4xLjEuMTY5NDQyNDQ5My4wLjAuMA... Ultima verifica: 11/09/2023.

— *UN APPROCCIO IAC CON CLOUD DEVELOPMENT KIT (CDK).* URL: https://www.zero12.it/un-approccio-iac-con-cloud-development-kit-cdk/?_gl=1*ld9qyg*_ga*NTQ1MjY4MTAyLjE2OTQ0MjQ0OTE.*_up*MQ..*_ga_EMNK7KNX8F*MTY5NDQyNDQ5MC4xLjEuMTY5NDQyNDQ5My4wLjAuMA..*_ga_F6C07J2PGB*MTY5NDQyNDQ5MC4xLjEuMTY5NDQyNDQ5My4wLjAuMA... Ultima verifica: 11/09/2023.

Serverless Framework. *Deliver software with radically less overhead.* URL: https://www.serverless.com/framework/docs. Ultima verifica: 11/09/2023.

Appendice A

Ottenere minori tempi di risposta dalle funzioni Lambda

Le funzioni Lambda basano il loro funzionamento sui *container*. Quando AWS istanzia una Lambda, questa viene incapsulata nel *container* che corrisponde al *runtime* specificato in fase di configurazione. A questo punto il sistema attende che venga generato l'evento che scatenerà l'esecuzione della funzione. Quando l'evento viene effettivamente generato si scatena un processo chiamato *cold start*. In questa fase il *container* viene effettivamente caricato nel sistema e messo in esecuzione. Successivamente gli verrà passato in *input* l'evento che ne ha scatenato l'esecuzione. Terminata l'esecuzione il *container* rimane in esecuzione per 15 minuti prima di essere fermato. Se nei 15 minuti nei quali il *container* rimane in esecuzione, un altro evento scatena l'esecuzione della stessa funzione il *timeout* viene impostato nuovamente a 15 minuti e viene eseguita la funzione. In questo caso si parla di *hot start*. Conoscendo questo meccanismo di funzionamento è possibile costruire delle funzioni che combinano più eventi diversi ma che ci si aspetta accadano in tempi ristretti per ottenere tempi di risposta medi più brevi. Questo approccio però viola il *single responsibility principle* che dovrebbe guidare lo sviluppo di una Lambda a regola d'arte. Se la violazione viene fatta consapevolmente e con cognizione di causa, il sistema risultante può effettivamente portare beneficio. C'è però da ricordare che più una Lambda è "grande", più memoria necessiterà, di conseguenza si ottiene una penalizzazione nei tempi di caricamento in caso di *cold start*, bisogna dunque fare delle valutazioni e *test* per ottenere un sistema ottimale che garantisca le performance desiderate.

Appendice B

Archiviazione S3 e Presigned URL

In AWS S3 come già visto si possono salvare dei file a lungo termine. Per poter ottenere tali file si possono utilizzare le API fornite ma queste non permettono ad un utente di scaricare il file tramite *browser*, ad esempio premendo su un *link*. Per risolvere questo problema, AWS ha introdotto il concetto di *presigned URL* (lett. URL pre-firmato). Questa soluzione consiste nel far generare al servizio un URL valido per un tempo limitato che codifica al proprio interno le seguenti informazioni:

- Il nome del *bucket* di destinazione;
- Il nome dell'oggetto su cui si lavora;
- Il metodo HTTP a cui il *server* risponde (GET per scaricare il file, PUT per caricare un file);
- Il tempo per cui l'URL risulta valido;
- L'*account* AWS utilizzato per generare l'URL.

Tutte queste informazioni sono necessarie al servizio per poter interagire con gli utenti, senza però dover cambiare configurazioni o politiche di accesso per un breve periodo o per singole operazioni.