Università di Catania

DANIELE COCUZZA 1000069220

Device Shop on Cloud

Relazione Progetto Sistemi Cloud

Proff: Giuseppe Pappalardo, Andrea Fornaia

ANNO ACCADEMICO 2023-2024

1 – Obiettivi

L’obiettivo del progetto è di sviluppare una piattaforma di e-commerce su cloud che consenta agli utenti di acquistare dispositivi per computer.

- **Visualizzazione degli Articoli Disponibili:** Gli utenti devono essere in grado di visualizzare una lista completa degli articoli disponibili per l'acquisto.
- **Aggiunta al Carrello:** Gli utenti devono poter aggiungere articoli al carrello per procedere con l'acquisto. Questa funzionalità facilita la selezione e la gestione dei prodotti da acquistare.
- **Ricerca dei dispositivi:** Gli utenti devono poter cercare i dispositivi utilizzando parole chiave per trovare rapidamente gli articoli di loro interesse. Questa funzionalità migliora l'usabilità e facilita la navigazione nel catalogo prodotti.
- **Avviso di Incompatibilità dei Dispositivi:** Il sistema deve avvisare gli utenti se hanno selezionato dispositivi che non sono compatibili tra loro, ad esempio, se un componente non è compatibile con un altro.
- **Memorizzazione dello Stato del Carrello:** Il sistema deve memorizzare lo stato del carrello dell'utente per consentire l'accesso e la gestione dei dati degli acquisti in sessioni future.

I microservizi che compongono l’applicazione saranno containerizzati e deployati su un cloud provider. La containerizzazione semplifica il processo di deploy, permettendo di creare ambienti isolati e replicabili, riducendo così il rischio di conflitti tra dipendenze e facilitando la gestione delle versioni e degli aggiornamenti.

Al fine di gestire in modo efficace il ciclo di vita del software, sarà implementata una pipeline CI/CD utilizzando le GitHub Actions. Le pipeline automatizzeranno il processo integrazione e deployment dei microservizi.

2 – Architettura e Tecnologie

Il progetto prevede la creazione di un e-commerce per dispositivi elettronici basato su tre microservizi sviluppati in Java utilizzando il framework Spring Boot e gestiti con Maven. I microservizi saranno containerizzati con Docker e orchestrati tramite Kubernetes, con distribuzione ed esecuzione su infrastruttura cloud.

Microservizi:

- Client: permette all'utente di inserire il nome con cui sarà salvata la sessione. Contiene il Cicuit Breaker tramite il quale effettuiamo le chiamate REST API. Il Cicuit Breaker ha un riferimento alla classe HttpRequest per mandare le richieste al server. Il client può visualizzare la lista dei componenti disponibili, può cercare un componente scrivendo una parola contenuta nel nome di esso, può acquistare un componente e visualizzare la lista dei suoi acquisti.
- Server: riceve le richieste dal Client. Il Server contiene i dati dei dispositivi in vendita, permette la ricerca dei dispositivi e verifica la compatibilità. In caso che il client chiede di visualizzare il suo carrello o chiede di acquistare un dispositivo il Server inoltra la richiesta al microservizio che si occupa di gestire la sessione.
- Sessione: si occupa di gestire la sessione dei client. Contiene una mappatura in cui ogni nome del client e associata la lista degli acquisti effettuati. Lo stato viene serializzato e salvato nel file system del microservizio così da salvare i dati dell'utente anche quando esso non è collegato.

Docker

La containerizzazione dei microservizi è stata realizzata creando immagini Docker per ciascuno dei tre servizi. I Dockerfile creano le immagini Docker per le applicazioni Java. Si parte da un ambiente Maven già configurato, copia i file del progetto all'interno dell'immagine, costruisce l'applicazione e alla fine esegue il packaging dell'applicazione Java. Questi Dockerfile contengono le istruzioni necessarie per installare le dipendenze, configurare l'ambiente e avviare i servizi. Le immagini Docker sono state testate localmente per garantire che ogni microservizio funzionasse correttamente all'interno del container. Ogni microservizio è stato racchiuso in un container Docker, che fornisce un ambiente isolato e coerente per l'esecuzione dell'applicazione. Questo ha incluso la verifica della compatibilità, la gestione delle dipendenze e la configurazione dell'ambiente di runtime. Per rendere le immagini Docker accessibili alla macchina virtuale EC2, è stato necessario effettuare il push delle immagini su Docker Hub, un registro pubblico di immagini Docker.

Kubernetes

Ho orchestrato i microservizi del progetto di e-commerce utilizzando Kubernetes, un sistema open-source progettato per automatizzare la gestione, il deployment e la scalabilità dei container. L'uso di Kubernetes è stato adoperato per garantire che i microservizi potessero essere gestiti in modo efficiente all'interno di un'infrastruttura cloud.

L'orchestrazione con Kubernetes ha permesso di suddividere i microservizi in deployment separati, ciascuno dei quali rappresenta un'istanza specifica del servizio. Ho configurato Kubernetes per gestire tre microservizi distinti: **Client**, **Server** e **Session**. Ogni microservizio è stato definito in modo tale da poter essere replicato, aggiornato e monitorato in modo indipendente.

Per ogni microservizio, ho configurato un **Service** in Kubernetes, che funge da astrazione per gestire il traffico in entrata verso i vari pod (le unità esecutive di Kubernetes che contengono i container). I Service hanno permesso di esporre i microservizi tramite un endpoint stabile, facilitando la comunicazione interna tra i componenti del sistema e l'accesso esterno dove necessario.

Sul fronte del networking, Kubernetes ha gestito la comunicazione tra i microservizi all'interno del cluster attraverso un sistema di networking interno, che utilizza nomi di servizio DNS per la risoluzione degli indirizzi. Questo ha eliminato la necessità di configurare indirizzi IP fissi per ogni microservizio, semplificando la gestione della rete e migliorando la resilienza del sistema.

Cloud AWS

Ho effettuato il deployment dell'applicazione di e-commerce su cloud utilizzando Amazon Web Services (AWS), sfruttando macchine virtuali EC2 di tipo **t3.medium**. Questo tipo di istanza EC2 è stato scelto per il suo equilibrio tra costo e prestazioni, offrendo una configurazione adeguata a supportare i carichi di lavoro del sistema. Le istanze **t3.medium** di AWS forniscono un mix ottimale di CPU, memoria e capacità di rete per applicazioni web e microservizi. Con 2 vCPU e 4 GB di memoria RAM, queste istanze sono ben equipaggiate per gestire l'esecuzione dei microservizi containerizzati. Le istanze T3 utilizzano un modello di CPU basato su crediti, che consente di ottenere elevate prestazioni durante i picchi di utilizzo, mantenendo allo stesso tempo i costi operativi contenuti durante i periodi di minor attività.

Ho creato una VPC dedicata per il progetto, scegliendo un intervallo di indirizzi IP privati adeguato a soddisfare le esigenze di comunicazione interna tra i vari microservizi. Questa rete è stata configurata per includere un indirizzo CIDR (Classless Inter-Domain Routing) che consente di assegnare indirizzi IP privati ai componenti della rete.

Ho configurato le tabelle di routing (Route Tables) per gestire il traffico di rete all'interno della VPC e verso l'esterno. Per consentire alle risorse della subnet privata di accedere a Internet (per esempio, per aggiornamenti software o comunicazioni con servizi esterni), ho configurato un Internet Gateway associato alla VPC.

CI/CD

L'applicazione è stata distribuita utilizzando un processo automatizzato attraverso GitHub Actions, che ha facilitato l'intero ciclo di integrazione e deployment continuo (CI/CD). Ogni volta che viene effettuato un push al repository, viene attivato un workflow che esegue la build e il push delle immagini Docker nel relativo repository su Docker Hub e il deployment dell'applicazione sulle istanze EC2. Questo processo automatizzato garantisce che le modifiche al codice siano

prontamente distribuite e rese operative senza la necessità di interventi manuali, riducendo al minimo il rischio di errori e assicurando un rapido time-to-market.

Al fine di permettere il deploy automatico ho aggiunto le seguenti directory:

ecommerce-cloud/

```
├── .github/
│   └── workflows/
        └── docker-publish.yml
│       └── ec2-deploy.yml
├── scripts/
│   └── deploy1.sh
│   └── deploy2.sh
```

Il file **docker-publish.yml** gestisce il processo di autenticazione su Docker Hub utilizzando le variabili segrete DOCKER_USERNAME e DOCKER_PASSWORD. Successivamente, costruisce le immagini Docker per i tre microservizi e le carica nel repository Docker Hub.

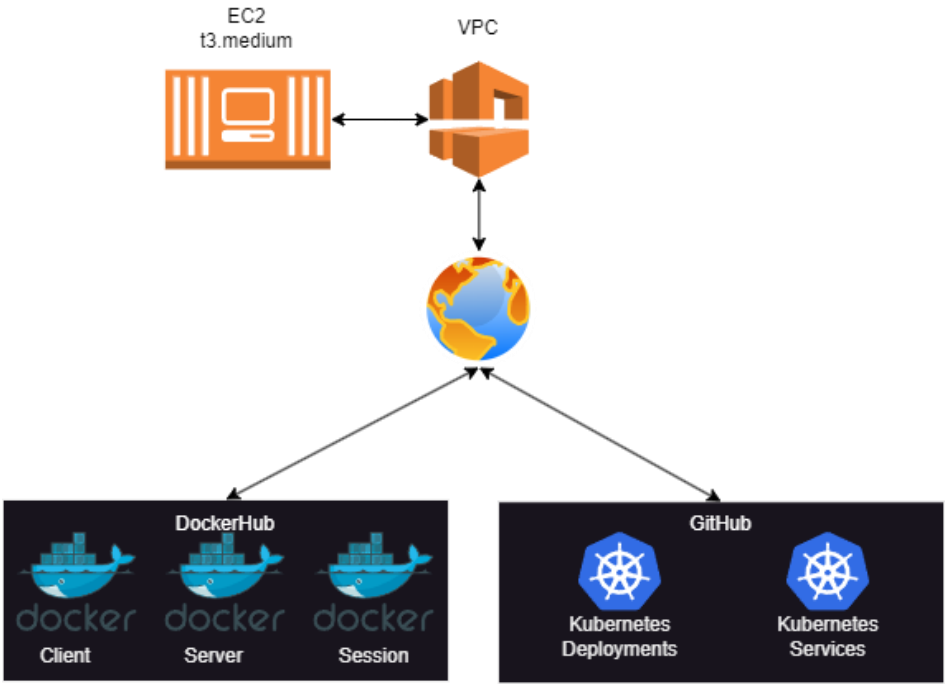
Il file **ec2-deploy.yaml** contiene il trigger che avvia l'automazione del processo di Continuous Deployment (CD). Questo trigger si attiva ogni volta che viene effettuato un push sul branch "main". Ogni modifica al codice che viene inviata al branch **main** del repository avvia automaticamente il workflow di deployment, garantendo che le nuove versioni del codice vengano distribuite sulla macchina EC2 designata. Dopo aver configurato l'autenticazione SSH, gli script vengono copiati dalla directory del repository locale alla directory **/home/ec2-user/** sulla macchina EC2 utilizzando scp. Successivamente, questi script vengono eseguiti in sequenza sulla macchina remota tramite SSH, garantendo che le operazioni di deployment vengano eseguite correttamente.

Il file **deploy1.sh** contiene le istruzioni necessarie per installare tutti i software e le dipendenze richieste per l'avvio del progetto. Questo include l'installazione di pacchetti, configurazioni di sistema, e altre operazioni preliminari indispensabili per il corretto funzionamento dell'applicazione.

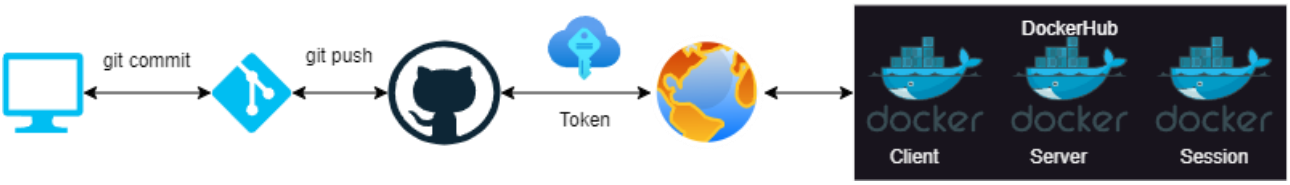
Il file **deploy2.sh**, invece, si occupa di avviare Minikube e di effettuare il deployment dell'applicazione su questo cluster Kubernetes. Questo passaggio comprende la creazione dei deployment e dei servizi necessari per eseguire l'applicazione su Minikube.

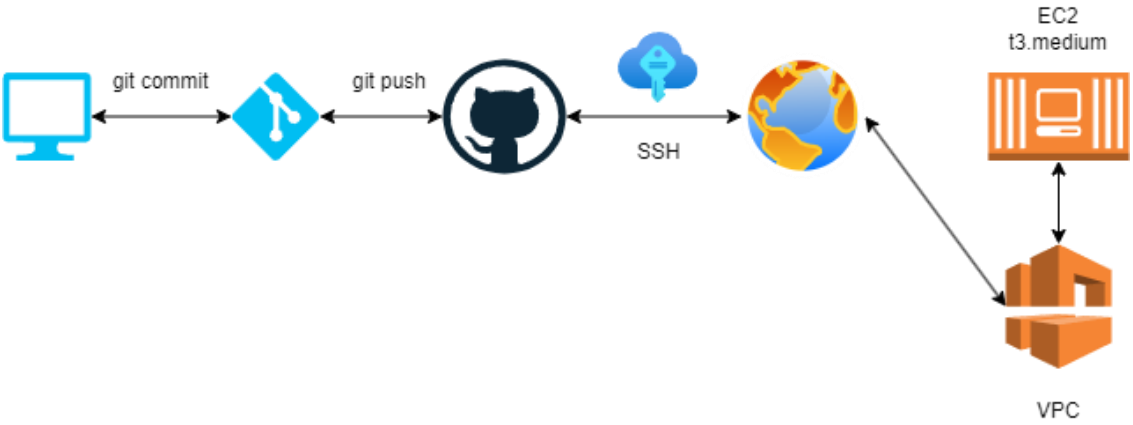
3 - Diagrammi

Localizzazione delle risorse



CI/CD Diagrams





4 – Conclusioni

Il progetto di e-commerce per dispositivi elettronici su cloud rappresenta un esempio concreto di come le tecnologie moderne possano essere utilizzate per sviluppare applicazioni scalabili, resilienti e gestibili. Attraverso l'utilizzo di microservizi sviluppati in Java e orchestrati tramite Kubernetes, si è riusciti a creare un'architettura distribuita che consente la gestione efficiente di diverse componenti dell'applicazione, mantenendo al contempo una forte separazione delle responsabilità.

La containerizzazione con Docker ha permesso di isolare le applicazioni e di standardizzare l'ambiente di esecuzione, semplificando il processo di distribuzione e rendendo le immagini facilmente accessibili tramite Docker Hub. L'integrazione con AWS, utilizzando macchine EC2 per l'esecuzione dei container, ha ulteriormente migliorato la scalabilità e l'affidabilità dell'infrastruttura.

Inoltre, l'implementazione della Virtual Private Cloud (VPC) su AWS ha garantito un ambiente sicuro e isolato, fondamentale per la protezione dei dati e delle comunicazioni tra i diversi microservizi. Il sistema è stato progettato per supportare un carico di lavoro dinamico, con la possibilità di scalare sia in verticale che in orizzontale a seconda delle esigenze.

Questo progetto dimostra la potenza e la flessibilità offerte dal cloud computing e dall'architettura a microservizi, fornendo una solida base per ulteriori sviluppi e ottimizzazioni. Le tecnologie utilizzate non solo garantiscono prestazioni elevate e disponibilità continua, ma offrono anche una grande facilità di gestione e monitoraggio dell'applicazione, caratteristiche essenziali per il successo di qualsiasi progetto software moderno.