RELAZIONE ASSIGNMENT 3 DI SOFTWARE ARCHITECTURES AND PLATFORM

Bedei Andrea(matricola 0001126957)
Bertuccioli Giacomo Leo(matricola 0001136879)
Notaro Fabio(matricola 0001126980)

28 Dicembre 2024

Indice

Capitolo 1

RISTRUTTURAZIONE DELL'ARCHITETTURA IN EVENT-DRIVEN MICROSERVICES

Per ristrutturare l'architettura del sistema in modo da sfruttare microservizi event-drive, siamo partiti dalla soluzione ottenuta alla fine dell'assignment 2, modificando le seguenti cose:

- introduzione di Apache Kafka come event broker tra i microservizi → i manager service dei servizi per le bici, le ride e per gli utenti contegono ora un CustomKafkaListener, affinchè possano captare gli eventi pubblicati sul broker Kafka principalmente dai managerImpl dei vari servizi o dal RidesExecutionVerticle e captati da ManagerImpl, RepositoryImpl (tranne per le ride siccome non esiste) e dal managerVerticle del microservizio ricevente. I vari componenti utilizzano gli eventi in questo modo: il ManagerImpl per aggiornare la sua copia locale dei dati, il RepositoryImpl per salvarli nell'event store ed il ManagerVerticle per inoltrarli ai web socket aperti dedicati alle GUI che si sono sottoscritte a tali eventi

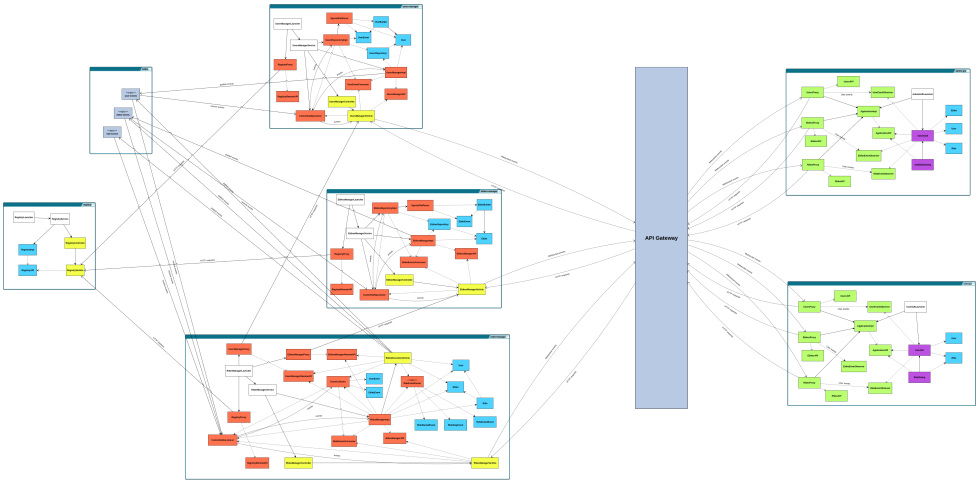
- applicazione del pattern event sourcing e conseguente sostituzione del database con un event store che contiene la sequenza di eventi di dominio accaduti → nota che adesso gli eventi salvati nel event store sono cumulativi, non rappresentano più lo stato corrente dell'oggetto ma la differenza rispetto allo stato precedente → è stata introdotta anche una classe SpecialFileParser, che manipola i file del database per ricostruire gli elementi di dominio contenuti e scrivere ulteriori eventi accaduti

- introduzione delle classi KafkaProducerFactory, KafkaConsumerFactory e CustomKafkaListener (è un wrapper di KafkaConsumer che consente ad altri oggetti, tipo i RepositoryImpl, i ManagerVerticle o l'EventConsumer, ovvero la copia locale dei dati delle bici e degli utenti usato dal RidesExecutionVerticle e dal RidesManagerImpl, di sottoscriversi agli eventi per ad esempio scrivere sull'event store)

- introduzione della classe EventCollector, situata dentro al microservizio delle ride → serve ad avere una copia locale, pur sempre aggiornata mediante eventi, di utenti e bici all'interno delle ride, in modo che il RidesManagerImpl e il RidesExecutionVerticle possano svolgere i loro compiti senza dover continuamente fare richieste HTTP agli altri microservizi → come effetto ulteriore è stato possibile smettere di usare alcune Future dentro le rides → avevamo valutato anche l'alternativa di usare gli eventi alla stregua delle richieste HTTP per richiedere i dati o venire aggiornati a livello delle singole entità (singola bici o singolo utente), ma per perseguire questa strada avremmo dovuto pensare ad un meccanismo per ignorare eventi non rivolti a me o includere l'id dell'ascoltatore nell'evento in modo da capire chi dev'essere il reale destinatario, ma abbiamo preferito quest'approccio più semplice, in cui l'EventCollector ascolta tutti gli eventi e mette a disposizione copie aggiornate di utenti e bici.

Si noti che la nostra soluzione utilizza sia eventi di tipo Entity Event (per utenti, ride-start e ebike) ma anche Keyed Event per gli eventi ride-step (per ride-step e ride-stop, che contengono l'informazioni riguardanti il movimento delle bici).

Per meglio comprendere come è stata modificata la struttura del sistema per costruire dei microservizi event-driven si faccia riferimento allo schema sotto riportato:



Lo schema è abbastanza simile a quello prodotto per l'assignment precedente, con le seguenti variazioni:

- gli eventi vengono ora propagati mediante il broker Kafka invece che con richieste HTTP e web socket (che comunque rimangono ancora per le GUI)
- nel microservizio Ride i dati di utenti e bici sono ottenuti per mezzo di eventi raccolti dall'EventCollector, invece di fare richieste HTTP esplicite.

Capitolo 2

CAMBIAMENTO DELL'INFRASTRUTTURA DI DEPLOYMENT DA DOCKER A KUBERNETES

La richiesta punto 2 della consegna richiedeva di passare da Docker a Kubernetes come infrastruttura di deployment.

Come visto durante il corso, Kubernetes è una piattaforma open-source per l'orchestrazione di container che aiuta a gestire, scalare e distribuire applicazioni containerizzate in modo automatizzato, consentendo agli sviluppatori di concentrarsi sullo sviluppo e agli operatori di semplificare la gestione dei carichi di lavoro.

Mentre Docker è uno strumento per creare e gestire container, ovvero unità isolate di software con tutte le dipendenze necessarie per funzionare, Kubernetes invece, si occupa della gestione di gruppi di container, orchestrandoli su più server (o nodi) e fornendo funzionalità come scalabilità automatica del numero di container, rilevamento e bilanciamento del carico ed autoriparazione (riavvio automatico dei container in caso di fallimento).
Ulteriore differenza tra i due risiede nel fatto che Docker può funzionare senza Kubernetes, ma Kubernetes utilizza strumenti come Docker (o container runtime alternativi) per gestire i container sottostanti.

I due concetti principali di Kubernetes che sono emersi durante il deployment sono:

- pod → è la più piccola unità di esecuzione in Kubernetes e rappresenta un gruppo di uno o più container che condividono stesso indirizzo IP,

porte, filesystem e ciclo di vita → i container all'interno di un Pod lavorano strettamente insieme e condividono risorse come se fossero parte dello stesso server

- cluster → è l'insieme di macchine (fisiche o virtuali) che lavorano insieme per eseguire e gestire i carichi di lavoro containerizzati.

Di seguito sono riportati i passi che abbiamo attuato per effettuare il deployment su Kubernetes:

- installazione di un ambiente Kubernetes in locale, fatto abilitando Kubernetes dalle impostazioni di Docker Desktop
- creazione del file **namespace.yaml** ed applicazione con **kubectl apply -f namespace.yaml** → un namespace serve a isolare e organizzare risorse all'interno di un cluster, separando logicamente applicazioni o ambienti diversi
- definizione della configmap nel file **configmap.yaml** e sua applicazione con **kubectl apply -f configmap.yaml** → una ConfigMap in Kubernetes è un oggetto che permette di memorizzare configurazioni non sensibili (come variabili di ambiente o file di configurazione) in modo separato dai container, facilitando la gestione e l'aggiornamento delle configurazioni senza modificare il codice o l'immagine del container
- per ognuno dei nostri servizi (e dunque ex container Docker) e anche per Kafka e Zookeeper, creazione del file di deployment (ad esempio **users-deployment.yaml**) e del file di service (ad esempio **users-service.yaml**) contenenti tutte le configurazioni necessarie → più nel dettaglio il file deployment.yaml specifica strategia di replica dei pod, gestione del ciclo di vita (Kubernetes si occupa di creare, aggiornare e mantenere i pod secondo lo stato desiderato), template dei pod (configurazione del container, ossia immagine, variabili d'ambiente, porte, volumi, ecc.), mentre il file service.yaml specifica accesso ai pod (permette di esporre i pod creati dal Deployment al resto del cluster), tipi di servizio (da noi è ClusterIP, il default, rende il servizio accessibile solo all'interno del cluster) → da questi file sono state tuttavia rimosse le sezioni readinessProbe e livenessProbe, che sono sì utili a capire se un container è pronto a ricevere traffico, se è ancora vivo o se necessita di essere riavviato, ma si è preferito rimuoverle per semplicità
- dopo aver preparato tutti i file di configurazione, per ognuno è stato necessario creare le risorse desiderate tramite i comandi **kubectl apply -f <nome-servizio>-deployment.yaml** e **kubectl apply -f <nome-servizio>-service.yaml**

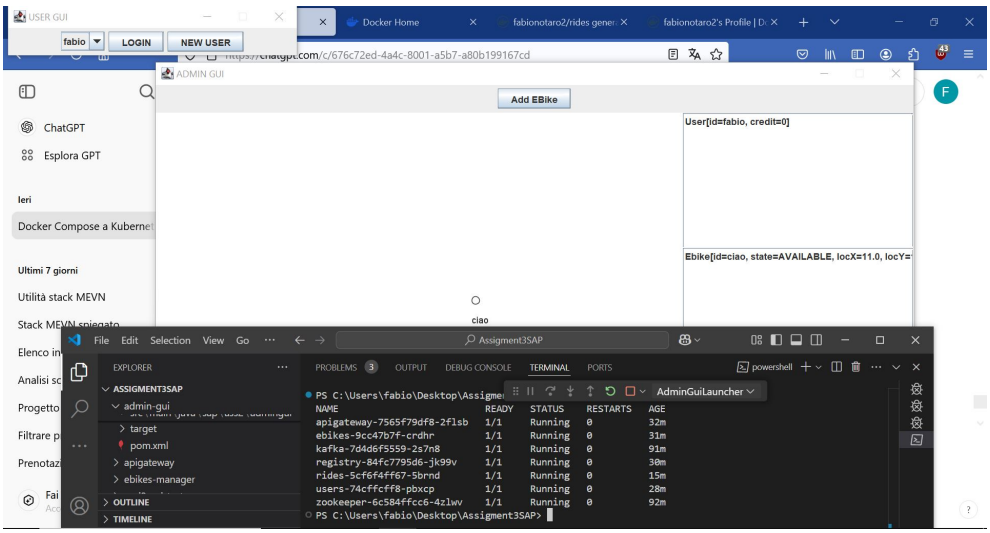
- testando se tutto è partito correttamente con **kubectl get pods -n ebikeapp** alcuni servizi non erano in stato Running ma in ImagePullBackOff siccome non riuscivano a fare la pull dell'immagine docker associata → per risolvere è stato necessario pubblicare sul profilo docker hub personale di un membro del gruppo le immagini dei container richieste, tramite i comandi
docker tag <nome-servizio>:latest fabionotaro2/<nome-servizio>:latest e **docker push fabionotaro2/<nome-servizio>:latest**, per ogni servizio che aveva questo problema →

```
PS C:\Users\fabio\Desktop\Assigment3SAP> kubectl get pods -n ebikeapp
NAME                          READY   STATUS    RESTARTS   AGE
apigateway-7565f79df8-2flsb   1/1     Running   0          17m
ebikes-9cc47b7f-crdhr         1/1     Running   0          17m
kafka-7d4d6f5559-2s7n8        1/1     Running   0          76m
registry-84fc7795d6-jk99v     1/1     Running   0          15m
rides-5cf6f4ff67-5brnd        1/1     Running   0          13s
users-74cffcff8-pbxcp         1/1     Running   0          13m
zookeeper-6c584ffcc6-4zlwv    1/1     Running   0          78m
PS C:\Users\fabio\Desktop\Assigment3SAP>
```

- a questo punto tutti i servizi erano in stato Running ma avviando la GUI dell'admin essa non andava correttamente perchè non riusciva a connettersi all'API Gateway, pertanto è stato necessario eseguire il port-forwarding tramite il comando **kubectl port-forward svc/apigateway 10000:10000 -n ebikeapp**, che inoltra la porta 10000 del servizio apigateway sulla macchina locale alla porta 10000, in modo da accedere al servizio API Gateway come se fosse in esecuzione localmente →

```
PS C:\Users\fabio\Desktop\Assigment3SAP> kubectl port-forward svc/apigateway 10000:10000 -n ebikeapp
Forwarding from 127.0.0.1:10000 -> 10000
Forwarding from [::1]:10000 -> 10000
Handling connection for 10000
Handling connection for 10000
Handling connection for 10000
```

- dopo il passo precedente le GUI riescono a contattare l'API Gateway e di conseguenza il cluster Kubernetes, pertanto le richieste inviate giungono correttamente a destinazione →


USER GUI
fabio
LOGIN
NEW USER
ADMIN GUI
Add EBike
User[id=fabio, credit=0]
Ebike[id=ciao, state=AVAILABLE, locX=11.0, locY=
ciao
ChatGPT
Esplora GPT
Ieri
Docker Compose a Kubernet
Ultimi 7 giorni
Utilità stack MEVN
Stack MEVN spiegato
File Edit Selection View Go
Assigment3SAP
EXPLORER
ASSIGMENT3SAP
admin-gui
target
pom.xml
apigateway
ebikes-manager
OUTLINE
TIMELINE
PROBLEMS 3
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
powershell
AdminGuiLauncher
PS C:\Users\fabio\Desktop\Assigme
NAME READY STATUS RESTARTS AGE
apigateway-7565f79df8-2flsb 1/1 Running 0 32m
ebikes-9cc47b7f-crdhr 1/1 Running 0 31m
kafka-7d4d6f5559-2s7n8 1/1 Running 0 91m
registry-84fc7795d6-jk99v 1/1 Running 0 30m
rides-5cf6f4ff67-5brnd 1/1 Running 0 15m
users-74cffcff8-pbxcp 1/1 Running 0 28m
zookeeper-6c584ffcc6-4zlwv 1/1 Running 0 92m
PS C:\Users\fabio\Desktop\Assigment3SAP>

Capitolo 3

ESTENSIONE BICI AUTONOME

In questo caso la soluzione di partenza è il punto precedente ma con deployment usando Docker, non Kubernetes, per motivi di familiarità.
Come pezzo core da implementare abbiamo deciso di fare in modo che le bici potessero raggiungere autonomamente l'utente che ha avviato la ride.

Per fare ciò sono state richieste alcune modifiche:

- dovendo la bici conoscere la posizione dell'utente che l'ha noleggiata (per raggiungerla), l'utente, che originariamente era descritto solo dal credito, necessita di avere anche la posizione all'interno del suo stato ed esporre delle API per muoversi
- introduzione del concetto di environment, che rappresenta un'astrazione che contiene tutti gli utenti e le bici, i quali vengono aggiornati in continuazione tramite eventi → in particolare, prima il RidesExecutionVerticle poteva conoscere i parametri di bici e utenti per mezzo di una richiesta HTTP GET esplicita agli altri microservizi, mentre ora l'ambiente viene notificato dall'event broker in caso di eventi/aggiornamenti di bici o utenti, aggiorna lo stato dell'entità di dominio interessata e lo mantiene e lo fornisce in caso di richieste al RidesExecutionVerticle
- a differenza di quanto si potrebbe immaginare, nella nostra soluzione gli agenti non sono le bici, in quanto esse sono entità puramente passive (conseguenza di come avevamo progettato l'assignment 2) → abbiamo perciò preferito fare in modo che sono le ride ad essere i nostri agenti, in quanto è il microservizio delle ride che si occupa del movimento delle biciclette → esistono, come prima, molteplici consumatori in ascolto sull'event loop del microservizio ride e uno di questi si occupa di inviare i messaggi step per avvisare tutti gli altri e anche se stesso di fare

il prossimo step delle rispettive ride, quindi ancora più precisamente possiamo dire che sono tali consumer ad essere gli agenti

- per conformarci a tale nuovo task, è stato necessario anche dividere ogni ride in due fasi ed esplicitare per ogni ride il suo stato → nella prima fase la bici si muove per raggiungere l'utente (per semplicità senza consumare batteria e credito), in questa fase la ride consulta di continuo l'environment e usa le informazioni estratte (posizione dell'utente) per aggiornare la sua direzione e spostarsi → raggiunto l'utente che ne ha fatto richiesta, la ride passa in modalità "normale", tradizionale, cioè quella che abbiamo implementato nelle scorse soluzioni, con l'unica differenza che anche la posizione dell'utente ora è legata a quella della bici (in questo caso, prima il microservizio delle ride aggiorna quello delle bici e quello degli utenti, di conseguenza tali modifiche verranno propagate anche all'environment siccome è in ascolto appositamente per questo)

- i nostri agenti svolgono dunque le classiche fasi di sense (richiesta all'environment della posizione dell'utente e della bicicletta), decide (calcolo della direzione opportuna) e act (modifica dei parametri della bicicletta ed eventualmente dell'utente)

- volendo classificare i nostri agenti, essi possono essere ricondotti ad agenti di tipo simple reflex agents, siccome non hanno uno stato interno ed ignorano le percept passate.