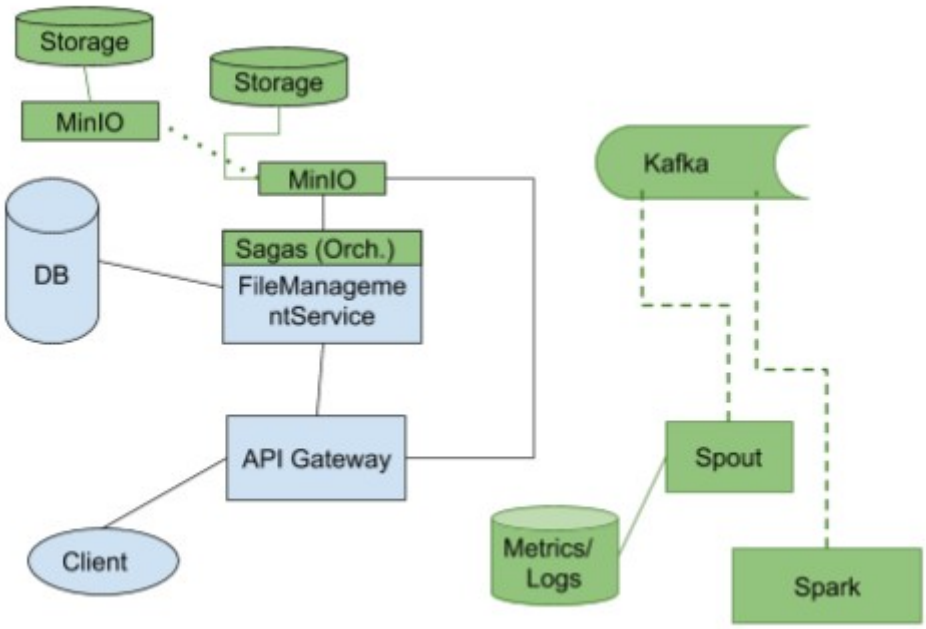
Object storage adapter

Coppolino Antonino O55000421
Labruna Mauro O55000429

Index


Storage
Storage
MinIO
MinIO
Kafka
DB
Sagas (Orch.)
FileManagemen
ntService
API Gateway
Spout
Client
Metrics/
Logs
Spark

Contenuto delle cartelle

homework_DSB:

è la cartella relativa al componente FileManagementService. All'interno della cartella è presente una sottodirectory **/files** nella quale sono caricati alcuni files di prova per testare l'applicazione. È stato utilizzato questo approccio perché non è stato implementato un client vero e proprio ma viene utilizzata l'estensione di Firefox "RESTED" per testare l'applicazione.

api_gateway:

la cartella è relativa al componente API_Gateway.

prometheus:

contiene il file prometheus.yaml il quale è necessario per configurare Prometheus per leggere le statistiche.

sparkConsumer:

contiene il codice relativo al consumer di Kafka. Il componente è stato scritto in linguaggio scala.
Il consumer preleva le statistiche dal broker Kafka che hanno come topic "metrics". Le statistiche vengono inviate a Kafka dal componente spout.

spout:

è la cartella relativa al producer di Kafka. Il componente è stato realizzato in Java.

storage_service:

è la cartella relativa al componente MinIO.

Funzionamento dei componenti

API Gateway:

Il componente si occupa di ricevere le richieste HTTP dal client, di smistarle verso il componente FileManagementService e di mostrare la risposta al client. L'API Gateway è stato configurato per avere un timeout massimo di risposta. Tale timeout è impostato mediante una proprietà di Spring Cloud Gateway

`(spring.cloud.gateway.httpclient.response-timeout)`

specificata nel file application.properties e tale tempo viene passato come variabile d'ambiente (nel nostro caso il timeout è di 3 secondi). Come richiesto dal progetto, l'API Gateway si occupa pure di fare un check sulla dimensione massima del file da caricare su MinIO. La dimensione viene passata come variabile d'ambiente e viene utilizzata all'interno della classe PayloadFilter. In tale classe viene fatto prima un check per vedere se il nome del file passato dal client è associato ad un file nella cartella homework_DSB/files ed in seguito viene fatto un check sulla dimensione del file da caricare. Se la dimensione supera la dimensione massima allora viene lanciata un'eccezione.

FileManagementService:

Il componente supporta una serie di API implementate come segue:

POST fms/register

Questa feature permette di registrare un nuovo utente, quindi, non necessita di alcuna forma di sicurezza.
La richiesta iniziale sarà indirizzata all'API Gateway, il quale grazie alla sua configurazione la inoltrerà a FileManagementService.
(riportiamo il codice solo per questa prima feature)

```
route(p -> p.path("/register/**")
        .filters(f -> f.rewritePath("/register", "/user/register")
                .filter(httpRequestInfo))
        .uri(uriConfiguration.getUrl()))
```

L'implementazione di questa feature è stata fatta mediante il metodo register di UserService del progetto homework_DSB

POST fms/

In questo caso è necessaria l'autenticazione, tutte le forme di sicurezza sono implementate nella classe AdapterConfiguration del progetto homework_DSB.

```
.antMatchers(HttpMethod.POST,"/record/
put").hasAnyAuthority("USER,ADMIN")
```

Se l'utente è gia registrato e l'autenticazione va a buon fine verrà inserito un nuovo record nel DB che conterrà il riferimento all'utente ed il filename.
La richiesta viene indirizzata verso FileManagementService con il path: record/put e viene quindi presa in gestione dal recordController.
Questa feature è stata implementata mediante il metodo saveRecord di RecordService.
Poichè non abbiamo un client vero e proprio il file viene preso da una cartella locale interna al progetto h*omework_DSB/*files. Insieme alla creazione dell'utente viene creato un bucket su MinIO associato all'utente creato.

POST fms/id

Anche in questo caso è necessaria l'autenticazione, inoltre, l'id deve essere associato ad un record già inserito nel DB, altrimenti l' applicazione tornerà un messaggio di errore 400.
La richiesta viene reindirizzata attraverso il path record/check/${id} e viene presa quindi in gestione dal RecordController.
Se l'id è presente nel DB e il filename è valido, il file sarà caricato su MinIO dentro il bucket dell'utente autenticato, se il file non esiste l'applicazione risponderà con un messaggio di errore.
Per come abbiamo configurato MinIO, ogni utente ha il suo bucket privato, il nome del bucket sarà uguale al nickname dell'utente che deve essere @Unique.
L'implementazione di questa feature è stata fatta mediante il metodo checkRecord di RecordService.

GET minio/id

La richiesta viene indirizzata verso il path /record/showRecord/${id} e viene presa in carico dal recordController.
Anche in questo caso è necessaria l'autenticazione, dobbiamo però distinguere due casi:

- utente autenticato ha ruolo USER: l'utente può accedere solo ai record (e quindi ai file) di cui e proprietario, nel caso in cui l'utente provasse ad accedere al file di un altro utente l'applicazione risponderebbe con un messaggio di errore.

Se il processo invece andasse a buon fine, all'utente sarebbe ritornato il link mediante il quale può scaricare il file da MinIO

- Utente autenticato ha ruolo ADMIN: l'utente può accedere a qualsiasi file presente su MinIO.

L'implementazione di questa feature è stata fatta mediante il metodo getRecord() di RecordService.

GET minio/

La richiesta viene reindirizzata verso il path /minio/files e viene gestita dal MinioController

Autenticazione necessaria, anche in questo caso dobbiamo fare la distinzione tra USER e ADMIN:

- utente con ruolo USER: l'applicazione tornerà la lista dei file di proprietà dell'utente
- utente con ruolo ADMIN: l'applicazione tornerà la lista di tutti i file presenti nel sistema.

L'implementazione di questa feature è stata fatta mediante il metodo getFileByUserRole() presente in MinioService nel progetto homework_DSB

DELETE minio/id

La richiesta viene reindirizzata verso il path /minio/deleteByUserRole/${id} e viene presa in carico dal MinioController.

È necessaria l'autenticazione ed in base a questa si possono fare due distinzioni:

- utente ruolo ADMIN: può cancellare qualsiasi file
- utente ruolo USER: può cancellare solo i file di sua proprietà

Questa feature è stata implementata mediante il metodo deleteByUserRole presente in MinioService,

Prometheus:

Grazie al file prometheus.yaml, Promethues viene configurato il modo tale da leggere le statistiche dall'API-Gateway ogni 5 secondi. Vedere la sezione Statistiche per approfondimenti.

MinIO:

MinIO è stato utilizzato come servizio di ObjectStorage. Il componente FileManagementService utilizza le API Java ufficiali di MinIO per interagire con il servizio.

Kafka:

Kafka viene utilizzato come broker message. Per permetterne il corretto funzionamento, Kafka è stato affiancato a ZooKeeper

Spout:

È il producer di Kafka.
spout preleva le statistiche da prometheus mediante la classe HttpConnectionConfig e le manipola in modo tale da filtrare le statistiche duplicate ed averle in formato JSON nella classe ResponseManipulation così da facilitare la realizzazione del consumer Kafka. Le statistiche, inoltre, prima di essere inviate, vengono ordinate in base al loro tempo di generazione grazie ad una TreeMap. Il tempo di generazione è stato inserito come tag all'interno di ogni statistica. Le statistiche vengono inviate a Kafka ogni 10 secondi.

sparkConsumer:

Il componente si occupa di prelevare le statistiche da Kafka ogni 10 secondi e di manipolarle come richiesto dalla consegna. Per analizzare le statistiche si fa uso di Spark. Poichè il DStream prelevato da Kafka è JSON compatibile, per accedere ai vari tag delle statistiche si è deciso di convertire il DStream in JSON mediante opportune librerie e di mappare i campi interessati come il tempo di risposta e il metodo della richiesta HTTP nella classe d'appoggio MetricsContext. Per calcolare la media dei tempi di risposta delle risposte HTTP si è creata la classe d'appoggio AvgClass, l'applicazione manderà un alert via email se il tempo di risposta medio delle operazioni di scrittura/lettura è maggiore del 20% rispetto a un qualsiasi valore medio di scrittura/lettura degli ultimi 5 batch. Nel dettaglio per confrontare i tempi di risposta con gli ultimi 5 batch si è pensato di implementare una lista di dimensione fissa (pari a 5) scartando volta per volta il valore in testa il quale corrisponde al valore più vecchio quando la lista è piena.

Statistiche

Per collezionare le statistiche delle varie richeste HTTP si è fatto uso di micrometer. La dipendenza è stata aggiunta al file pom.xml del modulo API Gateway. Per soddisfare le statistiche richieste nella consegna è stato implementato un custom filter in API Gateway mediante la classe HttpRequestInfo. In tale filtro viene istanziato un micrometer Timer, necessario per misurare il tempo di risposta alla richiesta http e ad esso sono stati aggiunti i tag: **router_uri**, **http_method**, **response**, **outcome, time** i quali indicano rispettivamente l'url a cui viene reindirizzata la richiesta, il metodo http utilizzato, la risposta del server, il risultato della risposta ed il tempo in cui viene generata la richiesta.
Il custom filter è stato poi aggiunto ad ogni route del Bean **myRoutes** nella classe ApiGatewayApplication.
Per monitorare le statistiche si è deciso di utilizzare il sistema di monitoraggio Prometheus poiché supportato da micrometer e facilmente integrabile con con Spring. Prometheus permette inoltre di fare delle query sulle statistiche collezionate per ottenere nuove statistiche personalizzate (es. il numero di richieste al secondo).

Creazione dei componenti e dockerizzazione

Sono stati creati i Dockefiles relativi al FileManagementService, API_Gateway, sparkConsumer ed spout. Per tutti gli altri componenti del progetto sono state utilizzate delle immagini già pronte su cui sono state effettuate le relative configurazioni per ottenere un corretto funzionamento. In particolare sono state utilizzate le immagini:

- **minio/minio** per MinIO
- **prom/prometheus** per Prometheus
- **wurstmeister/zookeeper** per Zookeeper
- **wurstmeister/kafka** per Kafka
- **mysql** per mysql

Porting su Kubernetes

Per fare il porting su k8s è stato utilizzato il software kompose. I file generati da kompose sono stati messi all'interno della cartella kompose_export.
Per il corretto funzionamento delle operazioni è stata montata tutta la directory principale nella directory /storage in minikube.

I servizi mysql e MinIO sono stati definiti come StatefulSets in quanto richiedono uno storage persistente.
Per MinIO, in particolare, sono state create 3 repliche (il valore può essere cambiato, il funzionamento non cambia) per aumentare la ridondanza dell'applicazione. Tutte le operazioni che riguardano quindi MinIO vengono eseguite su tutte le repliche. Per identificare gli indirizzi IP delle varie repliche in FileManagementService è stata creata la classe MinioDiscoveryService la quale si avvale della classe InetAddress per ottenere gli indirizzi di tutte le repliche. Nella classe MinioService viene quindi creato un minioClient relativo ad ogni replica e le operazioni vengono così eseguite su ogni singola replica.