Relazione Assignment #5 - From Monolith to Microservices

Giacomo Totaro
giacomo.totaro2@studio.unibo.it

Ottobre 2024

1 Struttura del progetto

Per realizzare un'architettura a microservizi ispirata al caso di studio relativo agli E-Scooter, sono stati sviluppati quattro servizi distinti:

- **Forwarding Service**, in esecuzione sulla porta 8080;
- **Rides Sevice**, in esecuzione sulla porta 8081;
- **Management Service**, in esecuzione sulla porto 8082;
- **User Service**, in esecuzione sulla porta 8888.

Ogni servizio è implementato come un modulo autonomo, accessibile tramite la rispettiva porta dedicata. L'entry point del progetto è rappresentato da *localhost:8080*, corrispondente all'URL del *Forwarding Service*, il quale si occupa di reindirizzare le richieste in base alla struttura dell'URL.

1.1 Forwarding Service

Questo modulo funge da *API Gateway* per l'intero sistema, implementato mediante Spring Boot e Java. È stata inoltre integrata la libreria *resilience4j* per la gestione del Circuit Breaker, il quale previene che un guasto in un servizio si propaghi ad altri servizi. Di seguito è riportata la struttura del modulo:

1. **ApiGateway:** Costituisce l'entry point dell'applicazione, definendo i *bean* associati a ciascuna rotta;
2. **CircuitBreakerConfiguration:** Contiene la configurazione del Circuit Breaker; 2.2

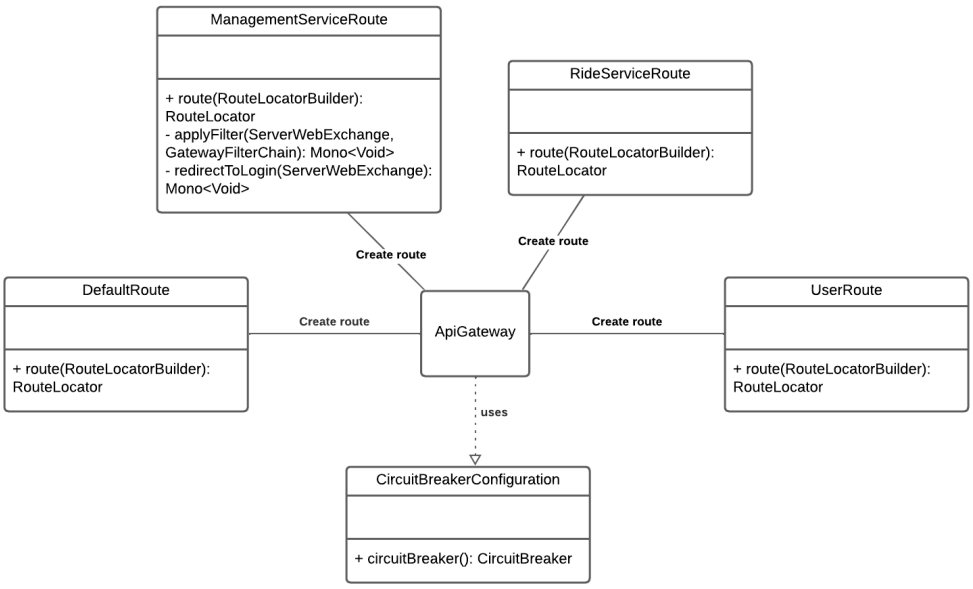

Figure 1: Forwarding Service Diagram

3. **Package Routes:** Comprende le diverse rotte gestite dall'API Gateway. Ogni classe all'interno di questo pacchetto implementa una funzione di routing che accetta un *RouteLocatorBuilder* e restituisce un *RouteLocator*, definendo il percorso e i filtri da applicare alle richieste.

Il routing è gestito da Spring Cloud Gateway, dove ogni rotta definisce un *bean RouteLocator* impiegato dal gateway per instradare le richieste in ingresso. Il metodo *uri* specifica la destinazione della rotta, corrispondente a un microservizio designato per gestire la richiesta. In caso di malfunzionamento del microservizio, interviene il Circuit Breaker.

1.2 Management Service

Questo modulo è stato implementato utilizzando Spring Boot e Java, strutturandosi come una tipica applicazione Spring Boot, con pacchetti distinti per *controllers*, *entità*, *repositories* e *configurazioni*.

- **ManagementServiceApp:** Rappresenta l'entry point dell'applicazione;
- **Package Controllers:** Include l'*Home Controller* e l'*E-Scooter Controller*. Il primo gestisce la richiesta *GET /api/management/dashboard* restituendo una pagina HTML, mentre il secondo si occupa di varie operazioni relative agli e-scooter, quali creazione, aggiornamento e visualizzazione dello stato;

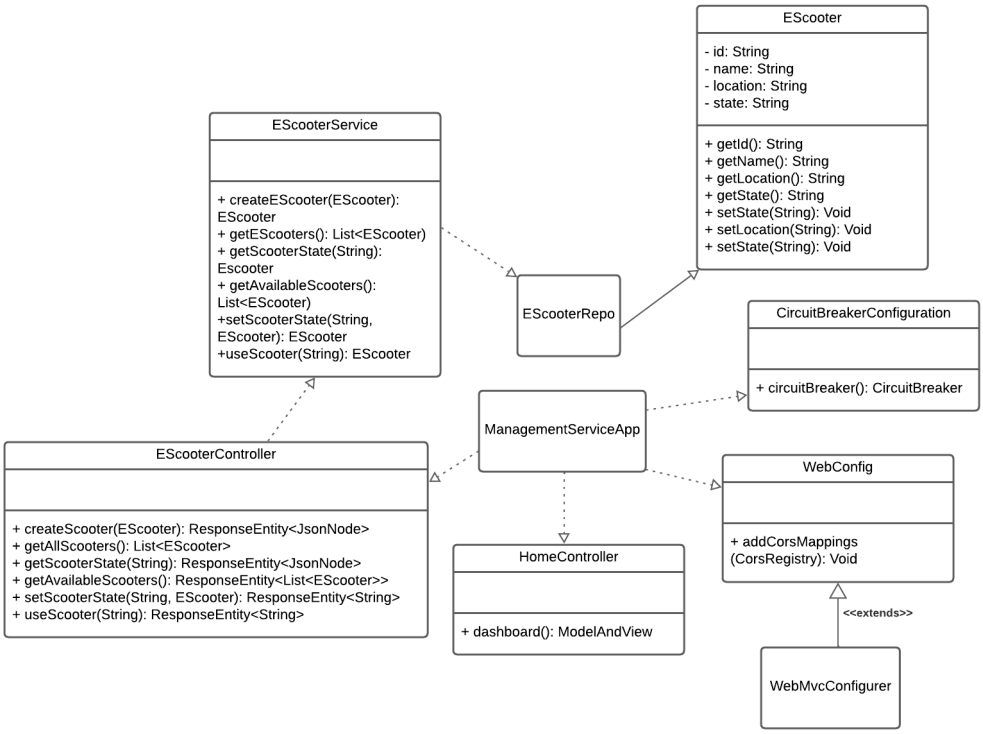

Figure 2: Management Service Diagram

- **Package Entities:** Contiene la classe *EScooter*, che rappresenta un'entità con id, nome e stato;
- **Package Repositories:** Comprende i file necessari per eseguire operazioni CRUD sulla collezione *eScooters* in MongoDB;
- **Package Config:** Include due classi: la prima dedicata alla configurazione del Circuit Breaker, mentre la seconda gestisce le impostazioni CORS, consentendo l'accesso a tutte le origini, intestazioni e metodi HTTP. Tale configurazione garantisce una maggiore flessibilità nelle interazioni tra client e server, facilitando le richieste provenienti da diverse origini.

1.3 Ride Service

Questo modulo è stato sviluppato utilizzando Vert.x e Scala per gestire le corse degli e-scooter.

- **RideServiceApp:** Costituisce l'entry point dell'applicazione, dove viene creata un'istanza di Vert.x e viene effettuata la distribuzione del verticle *RideServiceVerticle*;

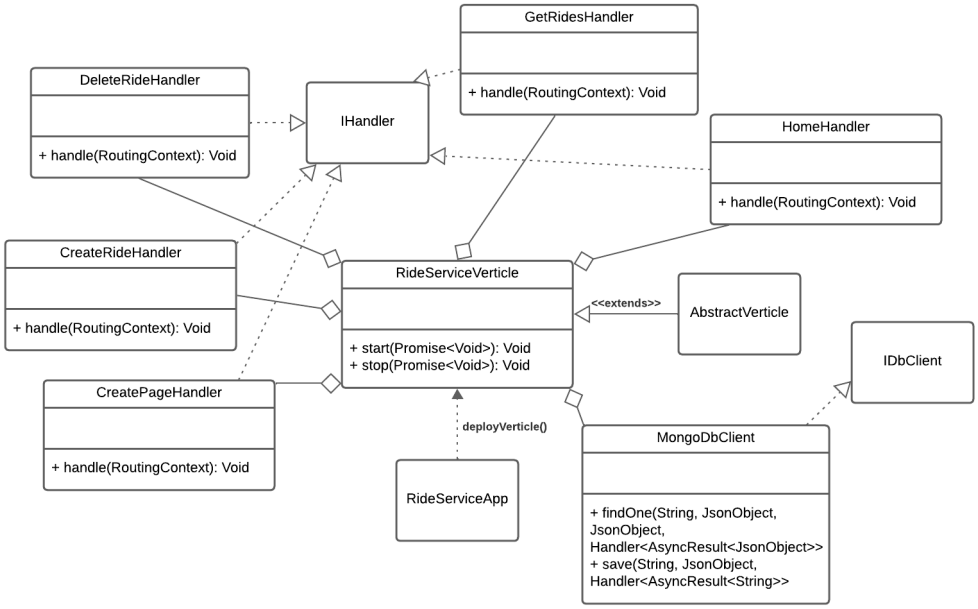

Figure 3: Ride Service Diagram

- **RideServiceVerticle:** Un verticle, componente fondamentale di Vert.x, che configura il server HTTP e le rotte per l'applicazione. Inizializza un *MongoDatabaseClient* e stabilisce le rotte utilizzando un'istanza di *Router*. Ogni rotta è associata a un gestore specifico responsabile dell'elaborazione delle richieste HTTP;
- **MongoDbClient:** Questa classe fornisce un'implementazione per interagire con il database MongoDB, offrendo metodi per cercare, salvare ed eliminare documenti dal database;
- **Handlers:** Comprende classi dedicate alla gestione di specifiche richieste HTTP;
- **Entities:** Include la classe *Ride*, che rappresenta l'utilizzo dell'e-scooter nel sistema, comprendendo proprietà quali id, località di inizio e fine, orario di inizio e di fine.

Il flusso dell'applicazione è il seguente:

- Quando un utente invia una richiesta, questa viene gestita dal *RideServiceVerticle*;
- Quest'ultimo utilizza il gestore appropriato per valutare la richiesta;
- Dopodichè, se necessario, utilizza *MongoDbClient* per interagire con il database MongoDB;
- Il *Ride Entity* viene utilizzato per rppresentare una 'ride' nell'applicazione e nel database.

1.4 User Service

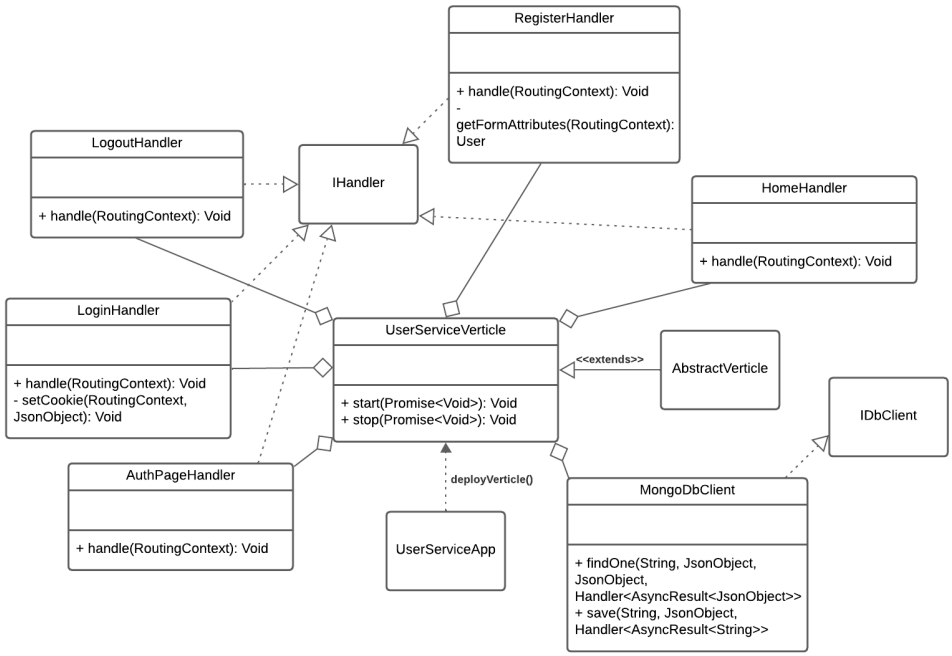

Figure 4: User Service Diagram

Questa applicazione è implementata in Java e Vert.x e fornisce funzionalità per la registrazione, il login e il logout.

- **UserServiceApp:** Rappresenta l'entry point dell'applicazione, dove avviene la distribuzione del verticle;
- **UserServiceVerticle:** In questo verticle viene inizializzato il database e vengono stabilite le diverse rotte, ognuna con il proprio gestore;
- **MongoDbClient e IDbClient:** Offrono un'interfaccia e la relativa implementazione per interagire con il database;
- **Handlers:** Comprende classi per la gestione delle richieste HTTP, come *LoginHandler* e *LogoutHandler*. Ogni gestore è responsabile dell'elaborazione di un tipo specifico di richiesta, come GET, POST o DELETE, e dell'esecuzione delle operazioni necessarie;
- **Entities:** Include la classe *User*, che rappresenta l'utente nel sistema, comprendente attributi quali nome, email, password e ruolo di manutentore.

2 Pattern

2.1 API Gateway

Nel presente progetto ho adottato il pattern *API Gateway* mediante l'uso della libreria *Spring Cloud Gateway*, la quale consente una configurazione minimale della logica necessaria per la realizzazione di un API Gateway essenziale. L'implementazione di tale pattern è stata effettuata tramite diverse classi *Route*, le quali gestiscono le richieste in base alla struttura dell'URL.

2.2 Circuit Breaker

Nel sistema è stato integrato il pattern *Circuit Breaker* per gestire i potenziali malfunzionamenti, consentendo la rilevazione e gestione dei guasti in modo tale da garantire la continuità operativa anche in presenza di errori. La gestione delle operazioni critiche è incapsulata nell'uso del metodo *executeSupplier* offerto dall'istanza del Circuit Breaker.

```
circuitBreaker.execute[Void]({ promise =>
  // Logica per eliminare la ride dal database
  databaseClient.deleteById("rides", rideId, { result =>
    if (result.succeeded()) {
      promise.complete()
    } else {
      promise.fail("Failed to delete ride")
    }
  })
}).onComplete({ ar =>
  if (ar.succeeded()) {
    routingContext.response().setStatusCode(200).putHeader(
       HttpHeaders.CONTENT_TYPE, "application/json").end(
       Json.encode("Ride deleted successfully"))
  } else {
    routingContext.response().setStatusCode(500).putHeader(
       HttpHeaders.CONTENT_TYPE, "application/json").end("
       Failed to delete ride")
  }
})
```

Nella mia classe *DeleteRideHandler*, il pattern *Circuit Breaker* è utilizzato per incapsulare le chiamate ai metodi del database. La seguente descrizione riassume il flusso di lavoro del Circuit Breaker:

1. Alla ricezione di una richiesta, questa viene gestita dal gestore specifico, il quale incapsula la chiamata al database attraverso il metodo *executeSupplier* del Circuit Breaker.

2. Se la chiamata al metodo del database ha esito positivo, il risultato viene restituito e il circuito rimane chiuso.
3. Qualora la chiamata al metodo generi un'eccezione, il Circuit Breaker registra il guasto.
4. Se il numero di fallimenti registrati supera una soglia predefinita in un determinato intervallo temporale, il Circuit Breaker si attiva e apre il circuito.
5. Una volta aperto il circuito, le ulteriori chiamate al database vengono bloccate e, in alternativa, il Circuit Breaker restituisce una risposta di fallback o genera un'eccezione.
6. Dopo un periodo di tempo prestabilito, il Circuit Breaker consente l'inoltro di un numero limitato di richieste di test. Se queste hanno successo, il circuito viene chiuso e il sistema riprende il normale funzionamento.

Segue un esempio di configurazione del Circuit Breaker:

```
@Configuration
public class CircuitBreakerConfiguration {

    @Bean
    public CircuitBreaker circuitBreaker() {
        CircuitBreakerConfig config = CircuitBreakerConfig.
           custom()
            .failureRateThreshold(50)
            .waitDurationInOpenState(Duration.ofMillis(1000)
               )
            .slidingWindowSize(2)
            .minimumNumberOfCalls(2)
            .build();

        return CircuitBreaker.of("circuitBreaker", config);
    }
}
```

- **failureRateThreshold**: La soglia di errore è impostata al 50%. Il Circuit Breaker si attiverà se il 50% o più delle chiamate fallisce nell'intervallo temporale definito.
- **slidingWindowSize**: La dimensione della finestra è impostata a 2 chiamate. Il Circuit Breaker valuterà gli ultimi 2 tentativi per calcolare la percentuale di fallimenti.
- **minimumNumberOfCalls**: Il numero minimo di chiamate necessario per aprire il Circuit Breaker è impostato a 2. Prima di raggiungere tale numero, il Circuit Breaker rimarrà chiuso.

- **waitDurationInOpenState**: Il Circuit Breaker resterà aperto per 1 secondo, durante il quale tutte le chiamate verranno rifiutate. Al termine di tale periodo, verrà eseguita una chiamata di test per verificare la disponibilità del servizio.

3 API RESTful

Nel mio progetto, ho adottato un approccio architetturale di tipo RESTful per la progettazione delle API, mirato a garantire flessibilità e scalabilità, tipiche dei sistemi distribuiti. Questo approccio si fonda sull'uso di operazioni HTTP standard (GET, POST, PUT, DELETE) per manipolare le risorse del sistema, in modo coerente e trasparente. Un esempio concreto di implementazione si trova nel microservizio **User Service**, che espone diverse API per la gestione delle operazioni relative agli utenti:

- **GET /api/users/dashboard:** Consente agli utenti autenticati di accedere alla propria dashboard, visualizzando informazioni personalizzate.
- **GET /api/users/login-form:** Fornisce la pagina di login per l'autenticazione degli utenti.
- **GET /api/users/register-form:** Restituisce la pagina di registrazione per nuovi utenti.
- **POST /api/users/auth/register:** Gestisce la registrazione di nuovi utenti, accettando i dati dal client e creando un nuovo account nel sistema.
- **POST /api/users/auth/login:** Permette agli utenti registrati di autenticarsi e creare una sessione attiva.
- **DELETE /api/users/auth/logout:** Effettua il logout degli utenti, terminando la sessione corrente.

Questo approccio RESTful permette di mantenere una chiara separazione delle responsabilità tra le varie operazioni di autenticazione e gestione degli utenti, garantendo al contempo la semplicità nell'estensione futura delle funzionalità.