Sap-Assigment02 Report

Shipping on the Air with Patterns

Bedeschi Federica
0001175655 federica.bedeschi4@studio.unibo.it

Pracucci Filippo
0001183555 filippo.pracucci@studio.unibo.it

Anno accademico 2025-2026

Indice

Quality Attributes

Abbiamo individuato un paio di **Quality Attribute (QA)** rappresentativi, dei quali abbiamo specificato gli scenari, testati come spiegato nella Sezione 4.4.
Il primo QA Scenario (vedi Listing 1.1) riguarda la responsiveness e ne specifica il requisito minimo in caso di overload del sistema.

Listing 1.1: QA Scenario – Responsiveness

```
Quality attribute scenario: Responsiveness

Feature: Small average response time in case of overload

when initiate 1000 concurrent requests
caused by 1000 users
occur in the system
operating in normal operation
then the system processes all requests
so that the average response time is < 100 milliseconds
```

Il secondo QA Scenario (vedi Listing 1.2) riguarda la gestione di partial failures e specifica di evitare l'invio di richieste inutili quando l'account service non è disponibile.

Listing 1.2: QA Scenario – Gestione partial failures

```
Quality attribute scenario: Handling partial failures

Feature: Avoid pointless requests in case of unavailability
   of account service

when more than 50% of total requests fail
caused by unavailability of account service
occur in the system
operating in normal operation
then the system makes the requests fail immediately without
   propagating to the account service
```

```
so that it lightens the workload avoiding pointless requests
```

Deployment

Per ogni microservizio è stato fatto il deploy su un singolo container **Docker**. Per il deploy dell'intero sistema, composto da vari microservizi, abbiamo utilizzato **Docker Compose**, che ne facilita la gestione consentendo la definizione di specifiche configurazioni per ogni servizio. Inoltre, ci permette di definire delle reti, quindi nel nostro caso abbiamo creato una rete condivisa tra tutti i servizi all'interno del sistema.

Patterns

3.1 API Gateway

Rispetto all'architettura dell'*Assignment-01* abbiamo aggiunto il microservizio dell'**API Gateway**, che agisce da unico entrypoint del sistema, nascondendo la composizione interna dei microservizi. Per fare questo espone una **REST API** lato client, ed internamente (tramite l'`APIGatewayController`) decide a quale microservizio inoltrare la richiesta sfruttando il relativo proxy (`AccountServiceProxy`, `LobbyServiceProxy` o `DeliveryServiceProxy`); ricevuta la risposta da quest'ultimo, risponde al client di conseguenza. La web socket che era presente tra il client e il delivery service viene sostituita da due web socket: una tra il client e l'API gateway e una tra l'API gateway e il delivery service. Questo permette di non esporre esternamente la gestione ad eventi utilizzata internamente.

3.2 Observability Patterns

3.2.1 Health check API

Osserviamo lo stato dell'API gateway tramite l'attributo `healthcheck` di Docker Compose, il quale effettua periodicamente richieste all'endpoint `api/v1/health` dell'API gateway controllando se il servizio è *healthy*. In caso contrario interviene un servizio aggiuntivo di `autoheal` tentando di riavviare l'API gateway. Abbiamo configurato l'endpoint sopracitato dell'API gateway in modo da ricevere le richieste dall'`healthcheck` e rispondere se *healthy* o meno. Inoltre, abbiamo aggiunto ad ogni servizio la policy di restart ad `always`, cosicché il relativo container venga riavviato in caso di fallimento finché non viene rimosso.

3.2.2 Application Metrics

Utilizziamo il servizio **Prometheus** per osservare le metriche definite, il quale viene aggiunto nel Docker Compose file e configurato nel file `prometheus.yml`. I due servizi che espongono metriche sono il delivery service e l'API gateway; per farlo devono esporre la porta configurata nel file `prometheus.yml` (ovvero relativamente 9400 e 9401). Il delivery service colleziona le *application level metrics*:

- `nTotalDeliveriesCreated`: numero totale di deliveries create;
- `nDeliveriesOnDelivery`: numero totale di deliveries in consegna;
- `nDeliveriesDelivered`: numero totale di deliveries consegnate;

L'API gateway colleziona le *infrastructure level metrics*:

- `nTotalNumberOfRESTRequests`: numero totale di richieste REST effettuate;
- `totalRequestResponseTime`: tempo di risposta totale delle richieste;
- `isAccountCircuitOpen`: indica se il circuito relativo al Circuit Breaker è aperto o chiuso;

Il delivery service utilizza l'*adapter* `PrometheusDeliveryServiceObserver` per esporre le metriche, implementando l'interfaccia `DeliveryServiceEventObserver` che a sua volta estende l'interfaccia `DeliveryObserver` (quest'ultima già creata per notificare gli eventi riguardanti le deliveries). Analogamente l'API gateway sfrutta l'*adapter* `PrometheusControllerObserver` per esporre le metriche, implementando l'interfaccia `ControllerObserver`, la quale viene utilizzata all'interno dell'`APIGatewayController`.

3.3 Event Sourcing

Applichiamo il pattern **Event Sourcing** al delivery service, in quanto è il servizio che si adatta meglio ad essere strutturato ad eventi, infatti avevamo già individuato diversi *domain events*, ai quali abbiamo aggiunto `DeliveryCreated`. Nello specifico persistiamo gli eventi al posto delle deliveries, in quanto *aggregate* principale del delivery service, sfruttando un **Event Store** basato su file. Di conseguenza per recuperare le deliveries già presenti nell'event store, creiamo una nuova istanza di ciascuna e applichiamo i relativi eventi in ordine.

3.4 Service Discovery

Il pattern **Service Discovery** è stato implementato a livello di deployment tramite Docker, il quale possiede *built-in* un **Service Registry** e utilizza un *DNS resolver* interno per localizzare dinamicamente i servizi, assegnando ad ognuno un *DNS name*.

3.5 Circuit Breaker

Abbiamo applicato il pattern **Circuit Breaker** all'interno dell'API gateway, in modo da risolvere parzialmente il problema legato alla possibilità di *partial failures* durante la comunicazione sincrona con altri servizi. Lo abbiamo implementato in ogni proxy tracciando il numero di richieste con successo e fallite, verificando se il tasso di richieste fallite supera una predeterminata soglia (50%), in tal caso il circuito viene aperto. In questa situazione l'API gateway risponde immediatamente al client con un errore, senza inoltrare la richiesta al servizio. Dal momento in cui il circuito viene aperto parte un timeout di 10 secondi, al termine del quale controlliamo se il servizio è tornato disponibile (inviando una richiesta all'endpoint relativo all'*health* del servizio), in caso positivo richiudiamo il circuito riprendendo il funzionamento normale.

Testing

4.1 Unit testing

Abbiamo realizzato **Unit tests** per ogni livello del delivery service:

- **domain**: abbiamo testato l'*aggregate* `Delivery` e l'entità `MutableDeliveryStatus`;
- **application**: abbiamo testato le principali funzionalità del `DeliveryService`;
- **infrastructure**: abbiamo testato il `DeliveryServiceController` appoggiandoci su un `DeliveryServiceMock`.

4.2 Integration testing

Come esempio di **Integration testing** abbiamo considerato il `DeliveryServiceProxy` sia nel lobby service che nell'API gateway. I due test possono essere considerati complementari, in quanto ognuno gestisce solo una parte delle richieste verso il delivery service e complessivamente coprono l'intera REST API esposta. Anche in questo caso abbiamo sfruttato un `DeliveryServiceMock`.

4.3 Component testing

Abbiamo implementato alcuni **Acceptance Tests** per testare il comportamento dei singoli servizi, sfruttando il tool *Cucumber*. Nello specifico abbiamo verificato le *user stories* per:

- **account service**: registrazione di un account e login;

- **delivery service**: creazione e tracciamento di una delivery.

Per ognuna abbiamo creato sia scenari di successo che di fallimento, dove necessario.

4.4 End-to-end testing

Per realizzare gli **end-to-end tests** abbiamo definito due *user journey* di esempio:

- *creazione delivery*: partendo dalla registrazione di un nuovo account, facendo il login e creando una nuova delivery;
- *tracciamento delivery*: partendo dal tracciamento di una delivery, ottenendo il relativo status per poi interrompere il tracciamento.

Inoltre, abbiamo implementato due test per verificare che i QA siano rispettati, ovvero un `PerformanceTest` per la *responsiveness* ed un `CircuitBreakerTest` per gestire le *partial failures*.
Tutti questi test sono realizzati mediante richieste esterne verso il sistema completo, a seguito del deploy con Docker.

Conclusioni

L'utilizzo di Docker e Docker Compose per fare il deploy dei microservizi ci ha permesso di realizzare un sistema più portabile, riproducibile e di aggiungere servizi esterni in maniera agile. Il deploy di ogni microservizio all'interno di un container separato consente un maggiore isolamento e quindi una minore dipendenza tra di essi. Nel caso in cui si presenti la necessità di scalare o espandere il sistema con altri microservizi, Docker facilita questo processo.
L'implementazione dei pattern ha migliorato il sistema per vari aspetti, per esempio l'API gateway ci ha permesso di nascondere la struttura interna del sistema, grazie all'esposizione di un singolo entrypoint, gli *observability patterns* abilitano il monitoraggio ed eventualmente l'*alerting* per consentire l'analisi e facilitare la gestione degli errori. Mentre gli altri patterns si concentrano nel rendere il sistema più robusto.
La creazione di un'ampia gamma di test, seguendo la *test pyramid*, ci ha permesso di aggiungere al controllo della correttezza delle singole funzionalità anche quella riguardante le comunicazioni tra i servizi e il sistema nella sua interezza. In particolare gli *end-to-end tests*, tramite *user journeys*, garantiscono che il comportamento atteso dall'utente nell'utilizzo del sistema sia rispettato.