

RELAZIONE ASSIGNMENT 2 DI SOFTWARE ARCHITECTURES AND PLATFORM

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

11 Novembre 2024

Indice

1	DOMAIN-DRIVEN DESIGN	3
1.1	Casi d'uso	3
1.2	Event storming	8
1.3	Pattern domain model	12
1.4	Ubiquitous language	13
1.5	Integration of Bounded Context Patterns	14
2	SVILUPPO	15
2.1	Come implementiamo le API?	15
2.2	Struttura del sistema	15
2.2.1	Architettura del servizio Registry	16
2.2.2	Architettura dei servizi Users e Bikes	17
2.2.3	Architettura del servizio Rides	19
2.2.4	Architettura delle GUI	20
3	PATTERN PER ARCHITETTURA A MICROSERVIZI	22
3.1	Pattern application-level	22
3.1.1	API Gateway	22
3.1.2	Circuit Breaker	23
3.1.3	Service Discovery	23
3.2	Pattern per deployment	24
3.3	Pattern per osservabilità	24
3.3.1	Health Check API	24
3.3.2	Application Metrics	25
3.3.3	Pattern di configurazione a runtime	27
4	PATTERN NON ADOTATI	28
4.1	Pattern saga	28
4.2	Pattern event sourcing	29
4.3	Pattern API composition	30
4.4	Pattern CQRS	30
4.5	Pattern di aggregazione di log	30
4.6	Pattern distributed tracing	31

4.7	Pattern exception tracking	31
4.8	Pattern audit logging	32
5	TESTING	33
5.1	Unit test	33
5.2	Integration test	33
5.3	Component test	34
5.4	End-to-end Test	34
5.4.1	Strategia User Journey	35
5.4.2	Esecuzione con Docker Compose	35
6	TESTTING DEI QUALITY ATTRIBUTE	37
6.1	Modularità	37
6.2	Performance degli endpoint	38
6.3	Disponibilità	39

Capitolo 1

DOMAIN-DRIVEN DESIGN

Il Domain-Driven Design è un approccio alla progettazione del software che si concentra sul dominio di business (ovvero sull'ambito del problema in cui il software realizzato opererà), rendendolo prioritario rispetto a considerazioni tecniche.

Riservando maggiore attenzione al dominio rispetto ai tecnicismi, ci si assicura che il prodotto software realizzato rifletta i bisogni, processi, concetti e comportamenti del contesto di dominio in cui verrà utilizzato.

Tra le varie tecniche di DDD affrontate a lezione, nel progetto si è analizzato nel dettaglio:

- casi d'uso
- event storming
- pattern domain model
- definizione di un ubiquitous language.

1.1 Casi d'uso

La definizione dei casi d'uso è un processo che coinvolge sviluppatori, analisti e stakeholder per capire cosa l'utente finale vorrebbe fare con il sistema ed in che modo vi interagirebbe.

In altre parole un caso d'uso può essere visto come uno scenario creato a partire d'un user goal, ovvero una sequenza di passi che descrivono un'interazione tra sistema e utente al fine di soddisfare un bisogno di quest'ultimo.

Come detto a lezione, non vi è un unico modo di descrivere i casi d'uso, tuttavia nel nostro progetto abbiamo preferito utilizzare la notazione elenco puntato piuttosto che il diagramma dei casi d'uso, in quanto abbiamo preferito esplorare questo approccio a noi meno familiare e che a nostro avviso

mette maggiormente in evidenza l'interazione instaurata tra utente e sistema piuttosto che un semplice insieme di possibili azioni svolte che sarebbero emerse con il diagramma UML dei casi d'uso.

Di seguito è riportato l'elenco puntato da noi sviluppato come scenari dei casi d'uso.

Partiamo dai casi d'uso relativi ai clienti del servizio.

1. Obiettivo: iscriversi al servizio

1.1 Il cliente apre l'applicazione

1.2 Il cliente sceglie un proprio username da utilizzare all'interno dell'applicazione

1.3 Il cliente preme il bottone per finalizzare l'iscrizione

[Estensioni]

1.2a L'username scelto è già utilizzato

1.2a.1 Il sistema avvisa l'utente del conflitto, invitandolo a cambiare username

1.2b.2 L'utente può riprovare l'iscrizione con un diverso username o uscire dall'applicazione

2. Obiettivo: loggarsi al servizio

2.1 Il cliente apre l'applicazione

2.2 Il cliente inserisce il proprio username

2.3 Il cliente accede alla sua area personale

[Estensioni]

2.2a L'username proposto dal cliente è incorretto

2.2a.1 Il sistema avvisa l'utente dell'errore

2.2b.2 Il sistema ripropone la procedura di login al cliente

3. Obiettivo: consultare il proprio credito

3.1 Il cliente accede all'applicazione

3.2 Il cliente accede alla sua area personale

3.3 Il sistema riporta a quanto ammonta il credito del cliente

3.4 Il cliente vede a quanto ammonta il suo credito

- 4. Obiettivo: consultare la lista di bici disponibili
 - 4.1 Il cliente accede all'applicazione
 - 4.2 Il cliente accede alla sua area personale
 - 4.3 Il sistema riporta la lista di bici disponibili, includendo per ogni bici l'id, il livello della batteria e la posizione
 - 4.4 Il cliente consulta la lista di bici
- 5. Obiettivo: ricaricare il proprio credito
 - 5.1 Il cliente va a consultare il suo credito
 - 5.2 Il cliente compila il campo dedicato alla ricarica del credito
 - 5.3 Il sistema risponde con la nuova quantità di credito
- [Estensioni]
 - 5.2a Il valore di ricarica proposto dal cliente è incorretto (cioè non positivo)
 - 5.2a.1 Il sistema avvisa l'utente dell'errore
 - 5.2a.2 Il sistema non aggiorna il credito
 - 5.2a.3 Il sistema ripropone la procedura di ricarica al cliente
- 6. Obiettivo: avviare un noleggio
 - 6.1 Il cliente consulta la lista di bici disponibili
 - 6.2 Il cliente seleziona una bici di suo interesse e preme il pulsante dedicato all'avvio del noleggio
 - 6.3 Il sistema verifica l'effettiva disponibilità della bici
 - 6.4 Il sistema verifica la presenza di credito sull'account del cliente
 - 6.5 Il sistema finalizza il noleggio e lo comunica all'utente
 - 6.6 Il sistema aggiorna in tempo reale il credito residuo del cliente
 - 6.7 Il sistema aggiorna in tempo reale la posizione e lo stato della bici noleggiata
- [Estensioni]
 - 6.3.a Il sistema vede che la bici selezionata dal cliente non è effettivamente disponibile
 - 6.3a.1 Il sistema avvisa l'utente del fallimento del tentativo

- 6.3a.2 Il sistema rimanda il cliente alla lista di bici
 - 6.4.a Il sistema vede che il cliente non ha credito
 - 6.4a.1 Il sistema avvisa l'utente del fallimento del tentativo
 - 6.4a.2 Il sistema rimanda il cliente alla lista di bici
 - 6.6.a Il sistema si accorge che il credito dell'utente si è esaurito durante il noleggio
 - 6.6a.1 Il sistema avvisa l'utente della fine del noleggio, includendo il motivo
 - 6.6a.2 Il sistema termina il noleggio e dunque la decurtazione del credito del cliente e l'aggiornamento dei parametri della bici
 - 6.7.a Il sistema si accorge che il livello di batteria della bici si è esaurito durante il noleggio
 - 6.7a.1 Il sistema avvisa l'utente della fine del noleggio, includendo il motivo
 - 6.7a.2 Il sistema termina il noleggio e dunque la decurtazione del credito del cliente e l'aggiornamento dei parametri della bici
 - 7. Obiettivo: terminare un noleggio
 - 7.1 Il cliente visualizza il noleggio in corso
 - 7.2 Il cliente preme il bottone per terminare il noleggio
 - 7.3 Il sistema termina il noleggio e dunque la decurtazione del credito del cliente e l'aggiornamento dei parametri della bici
- Passiamo ora ai casi d'uso relativi agli amministratori del sistema.
- 8. Obiettivo: consultare l'elenco di bici attive
 - 8.1 L'amministratore accede all'applicazione
 - 8.2 Il sistema riporta la lista di tutte le bici presenti, includendo per ogni bici l'id, il livello della batteria e la posizione
 - 8.3 L'amministratore consulta la lista di bici
 - 9. Obiettivo: aggiungere una bici
 - 9.1 L'amministratore accede all'applicazione
 - 9.2 L'amministratore preme il pulsante per aggiungere una nuova bici

- 9.3 L'amministratore riempie i campi per le informazioni della bici, includendo id e posizione
- 9.4 Il sistema registra la nuova bici
- 9.5 Il sistema comunica all'amministratore il successo dell'operazione
- [Estensioni]
- 9.3.a L'amministratore specifica un id invalido o già utilizzato
 - 9.3a.1 Il sistema notifica l'amministratore dell'errore
 - 9.3a.2 Il sistema riporta l'amministratore a cambiare l'id della bici che intende aggiungere
- 10. Obiettivo: rimuovere una bici
 - 10.1 L'amministratore accede all'applicazione
 - 10.2 Il sistema riporta la lista di tutte le bici presenti, includendo per ogni bici l'id, il livello della batteria e la posizione
 - 10.3 L'amministratore seleziona la bici da rimuovere dalla lista
 - 10.4 L'amministratore preme il pulsante dedicato alla rimozione della bici
 - 10.5 Il sistema avvisa l'utente del successo dell'operazione
 - [Estensioni]
 - 10.4a L'amministratore tenta di eliminare una bici in uso
 - 10.4a.1 Il sistema avvisa l'utente dell'impossibilità di completare la rimozione della bici
- 11. Obiettivo: consultare l'elenco di clienti iscritti
 - 11.1 L'amministratore accede all'applicazione
 - 11.2 Il sistema riporta la lista di clienti iscritti, includendo per ognuno l'username, il credito
 - 11.3 l'amministratore consulta la lista di clienti fornita
- 12. Obiettivo: consultare la posizione delle bici in tempo reale
 - 12.1 L'amministratore accede all'applicazione
 - 12.2 Il sistema mostra graficamente l'id e le posizioni delle biciclette nello spazio in tempo reale
 - 12.3 L'amministratore consulta i movimenti delle bici

1.2 Event storming

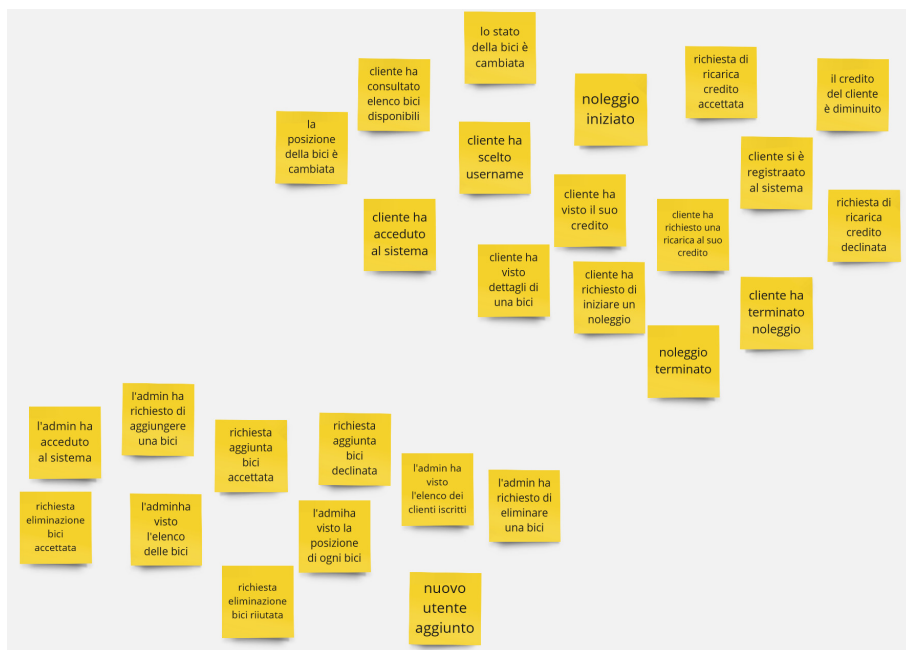
L'event storming è una tecnica utilizzata nel DDD per facilitare l'esplorazione collaborativa dei processi che governano il dominio attraverso dei post-it ed una lavagna collaborativa.

In altre parole, è una tecnica che aiuta analisti e sviluppatori a comprendere, esplorare e comunicare le dinamiche del dominio concentrandosi sugli eventi che caratterizzano il comportamento del sistema.

Si noti che progredendo nelle varie fasi che compongono l'event storming, gli eventi sono stati aggiunti, aggregati, rimossi tra i vari step.

I passi che compongono questa attività sono:

- esplorazione non strutturata → si raccolgono e collezionano gli eventi di dominio rilevanti (post-it arancioni) →



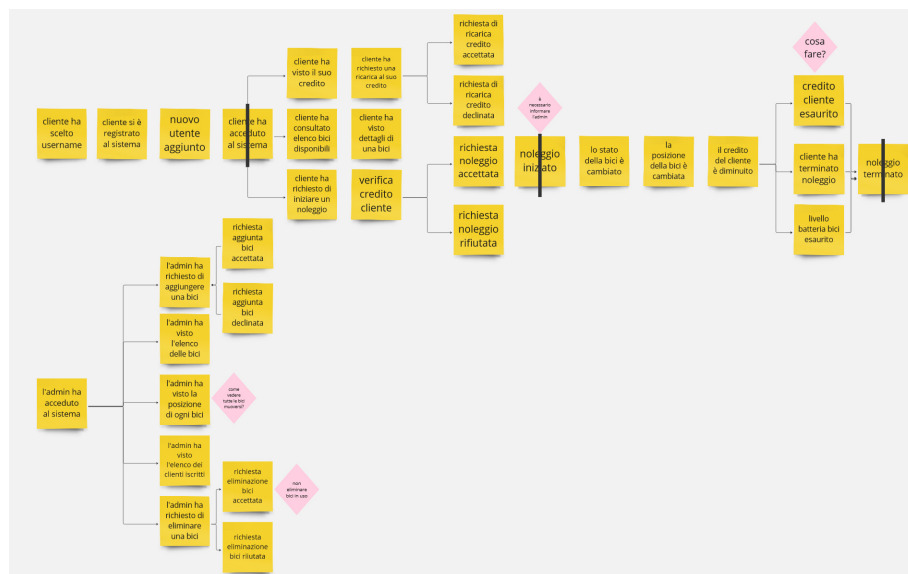
- timeline → si ordinano gli eventi del dominio individuati →



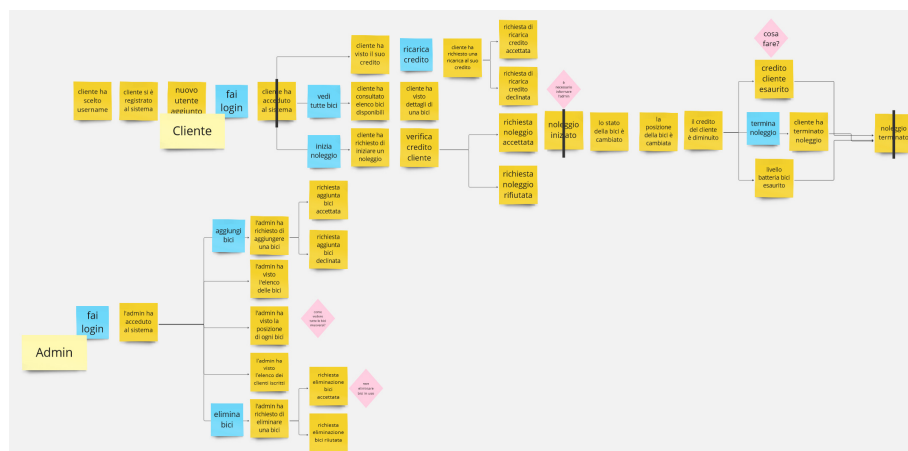
- pain points → si identificano punti del processo che richiedono attenzione particolare (postit romboidali rosa) →



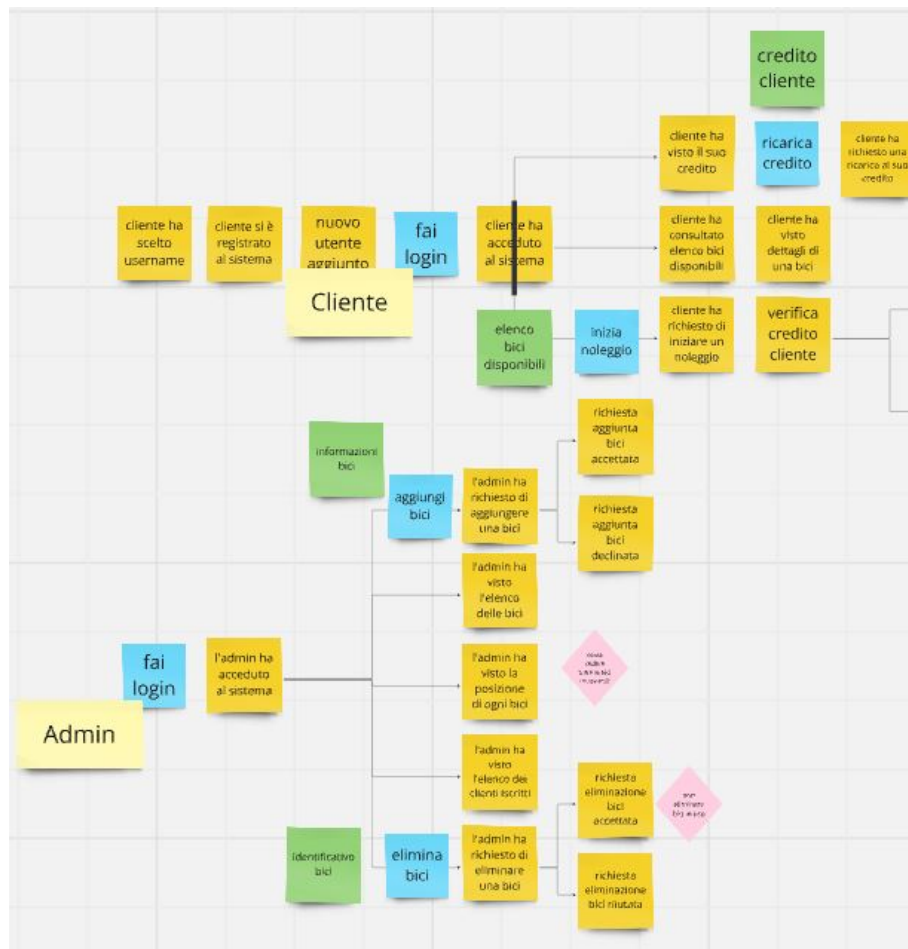
- eventi pivotali → si evidenziano gli eventi che indicano un cambiamento di fase/stato (barre verticali) →



- comandi → si aggiungono comandi (cosa ha scatenato un evento, postit blu) e attori (chi ha eseguito un comando, postit gialli) →



- policies → si identificano gli eventi che causano l'esecuzione di comandi → nessuna policy nel nostro event storming
- read model → si identificano i dati del dominio che l'attore usa e considera per eseguire un comando (postit verdi) →



- sistemi esterni → si evidenziano i sistemi esterni che possono eseguire comandi o essere notificati da eventi → non è emerso nessun sistema esterno
- aggregates → si organizzano i concetti correlati in aggregates
- bounded context → si cercano aggregati tra loro correlati e si raggruppano per creare bounded context.

L'event storming finale è il seguente:

- repository
- moduli.

Il risultato di questa fase è visibile direttamente nel codice, in cui tutti questi componenti sono stati opportunamente indicati ed etichettati tramite l'utilizzo di interfacce apposite (similmente a quanto visto in laboratorio).

1.4 Ubiquitous language

Le fasi precedenti ci hanno consentito di formalizzare un cosiddetto ubiquitous language, ovvero un linguaggio comune e condiviso (sia dagli sviluppatori che dagli esperti del dominio), utilizzato sia in fase di design che di sviluppo, con cui si identificano dei concetti chiave tramite termini condivisi, promuovendo di fatto la consistenza e la disambiguazione nel tentativo di limitare le incomprensioni.

Il glossario sotto riportato sintetizza i termini e concetti principali emergenti:

TERMINE	DEFINIZIONE	SINONIMI
Cliente	Utente che utilizza il servizio per noleggiare le ebike	Customer
Ebike	Bicicletta elettrica gestita dal sistema e noleggiabile dai clienti attraverso l'applicazione	Bici, bicicletta, bike, bici elettrica
Ride	Periodo di noleggio che termina quando il cliente restituisce la bici	Corsa, noleggio
Admin	Operatore del sistema che gestisce e controlla l'infrastruttura e lo stato delle ebike	Amministratore
Username	Identificativo scelto dal cliente durante la registrazione ed usato per accedere al sistema	Nome utente, id
Credito	Somma di denaro disponibile nell'account del cliente, utilizzata per i pagamenti dei noleggi delle ebike	
Inizia noleggio	Azione con cui un cliente avvia la richiesta di noleggio di una bicicletta	
Richiesta Noleggio	Stato che indica che la richiesta di noleggio è stata approvata o rifiutata	
Battery level	Livello percentuale della batteria di una bici	Batteria, livello batteria
Stato bici	Stato che rappresenta la situazione in cui si trova la bici (disponibile, in manutenzione, in uso o disabilitata)	Ebike state

Tabella 1.1: tabella con termini, definizioni e sinonimi

Si noti che anche a livello di codice si è cercato di mantenere l'utilizzo di questi termini.

1.5 Integration of Bounded Context Patterns

Sebbene ogni bounded context sia stato mappato in un microservizio distinto, alcuni bounded context non sono completamente indipendenti, poiché i componenti devono interagire tra loro.

Abbiamo scelto di adottare il modello di **Cooperation**.

Sebbene attualmente ci sia un solo team che si occupa dell'intero sistema, l'intento è che, qualora diversi team fossero assegnati a ciascun microservizio, sia presente un meccanismo di notifica per le modifiche alle interfacce tra i team cooperanti, implementando così il **Partnership pattern**.

Capitolo 2

SVILUPPO

2.1 Come implementiamo le API?

Principalmente implementiamo le API in modo asincrono (il client può proseguire nel suo flusso di controllo mentre attende la risposta ad una sua richiesta), tramite HTTP.

Indipendentemente dal meccanismo di inter process communication, abbiamo definito le API di ogni servizio utilizzando l'**API-First Approach**, nel quale si parte nel progetto definendo prima le API e poi tutto il resto.

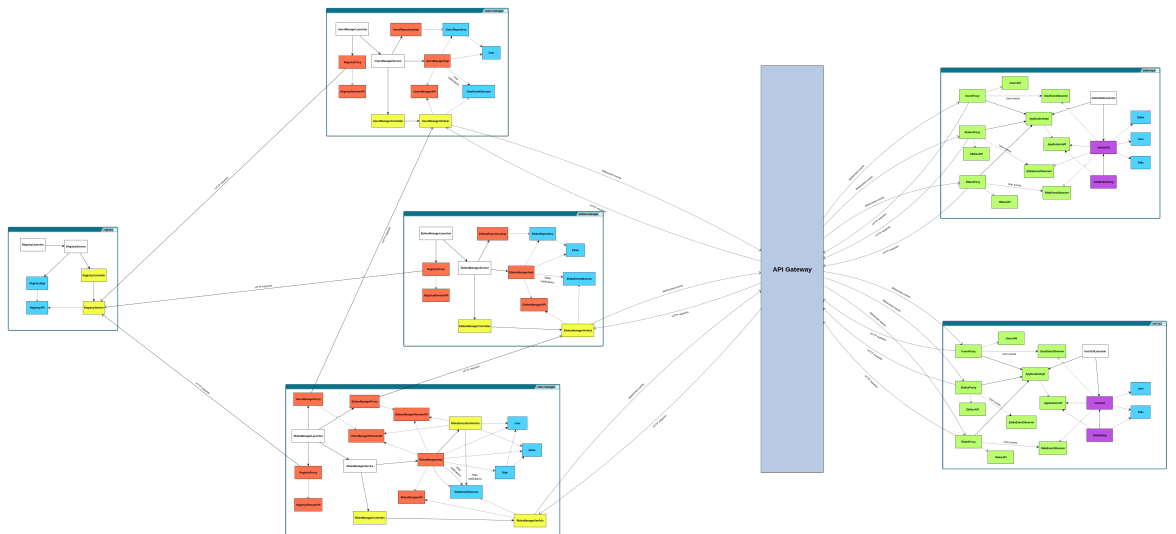
2.2 Struttura del sistema

Nella presenta sezione viene descritto come sono stati implementati e strutturati i microservizi e le GUI, insieme ai loro modi di interazione.

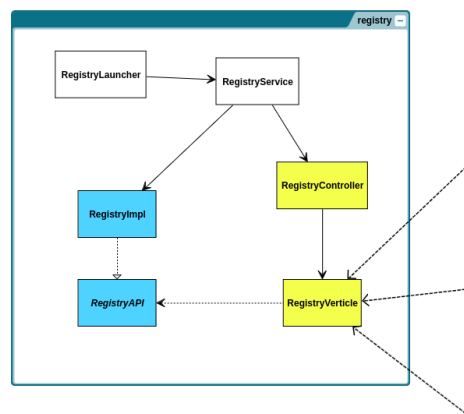
La figura sottostante mostra la struttura globale dell'intero sistema. Si noti che nello schema sopra e anche nei prossimi, è stata adottata la seguente convenzione sui colori:

- blu = domain layer
- arancione = application layer
- giallo = infrastructure layer
- verde = libreria (tutto ciò che non è nè GUI nè dominio)
- viola = GUI
- bianco = startup (utilities per l'avvio dei servizi).

Le sottosezioni che seguono si concentrano sull'architettura interna dei singoli microservizi.



2.2.1 Architettura del servizio Registry

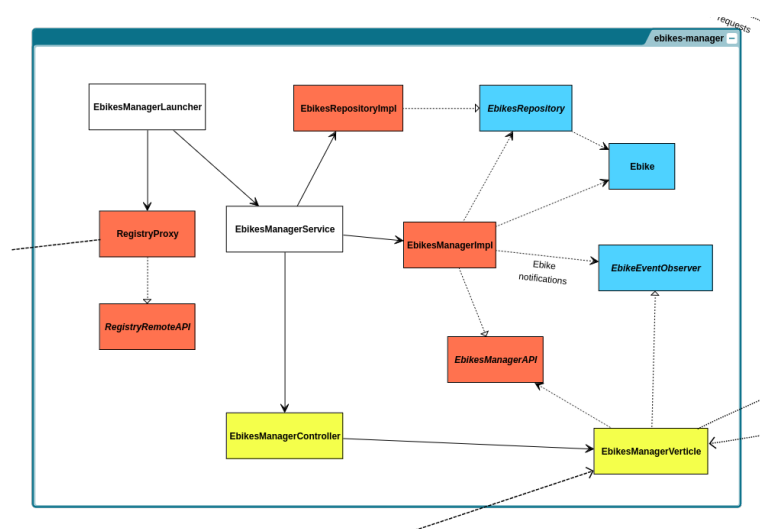


Questo microservizio rappresenta la nostra implementazione del pattern Service Discovery, ossia è il componente che viene interrogato da tutti gli altri servizi per registrarsi e/o trovare gli indirizzi desiderati. Esso è composto da:

- RegistryLauncher ha semplicemente il compito di creare il Registry-Service

- RegistryService ha il semplice compito di creare il RegistryController (responsabile dell'avvio del RegistryVerticle) e RegistryImpl (è il registro vero e proprio, infatti contiene i riferimenti ai servizi che lo hanno contattato)
- RegistryVerticle è il server HTTP per gestire le richieste che sopraggiungono al servizio → ogni richiesta HTTP viene pubblicata sull'event bus, a cui è sottoscritto RegistryImpl, il quale soddisfa la richiesta in modo che il verticle possa rispondere tramite una risposta HTTP al servizio esterno richiedente.

2.2.2 Architettura dei servizi Users e Bikes

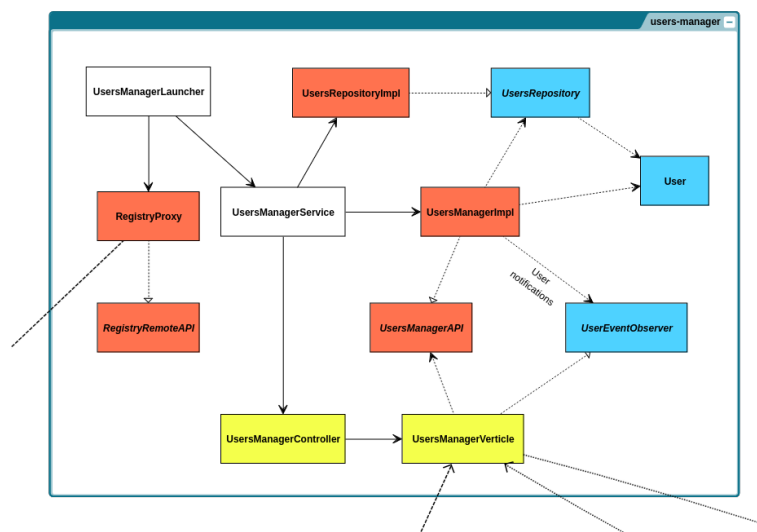


Il microservizio dedicato alle biciclette è invece composto da:

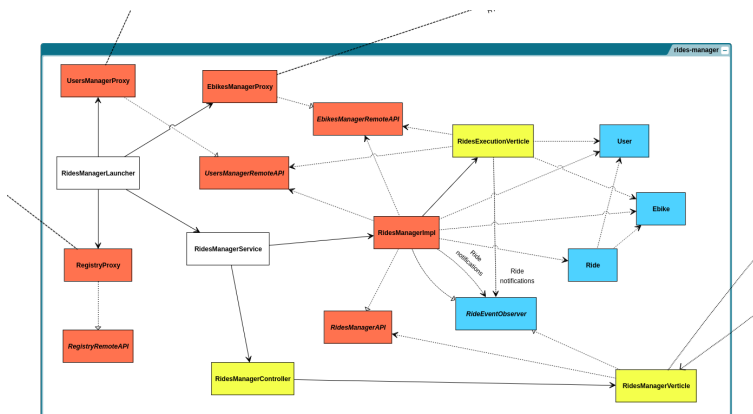
- EbikesManagerLaucher istanzia RegistryProxy (client HTTP per interagire con il servizio Registry) e EbikesManagerService
- EbikesManagerService ha il solo compito di istanziare EbikesRepositoryImpl, EbikesManagerImpl ed EbikesManagerController
- EbikesRepositoryImpl è l'implementazione delle operazioni eseguibili sul repository delle biciclette
- EbikesManagerImpl è il cuore del servizio, responsabile di tutte le operazioni sulle biciclette, accetta sottoscrizioni e propaga all'EbikesManagerVerticle (e indirettamente all'AdminGUI) gli eventi rilevanti accaduti alle biciclette

- EbikesManagerController ha il solo compito di creare l'EbikesManagerVerticle
- EbikesManagerVerticle è il server HTTP che gestisce non solo le richieste (provenienti dalle GUI e dal servizio delle rides principalmente) ma anche le sottoscrizioni ad eventi provenienti dalle GUI, alle quali il verticle sarà collegato mediante un WebSocket → più dettagliatamente, quando un esterno (tipo una GUI) si sottoscrive, il verticle tiene aperto un web socket dedicato alla comunicazione e crea un consumer per gli eventi di modifica delle ebike, che ascolta sull'event bus e quando il manager avvisa il verticle del nuovo evento, esso pubblica un messaggio appropriato sull'event bus, che verrà poi ricevuto dai consumer, i quali provvederanno a propagarlo sul WebSocket → per quanto riguarda la chiusura del WebSocket, esso attende messaggi "unsubscribe" per chiudersi
- EbikeEventObserver è l'interfaccia degli ascoltatori (in questo caso solo il EbikesMnsngerVerticle) per gli eventi sull'event bus.

Il microservizio dedicato agli utenti risulta del tutto analogo al precedente:



2.2.3 Architettura del servizio Rides



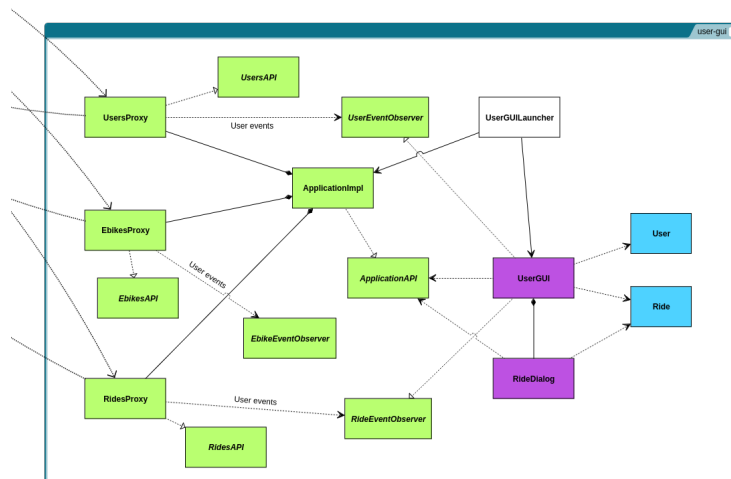
Esso è composto da:

- RidesManagerLauncher è il responsabile dell'avvio del servizio, in quanto crea non solo il RidesManagerService ma anche un proxy per il Registry, uno per gli utenti e uno per le bici (ciascun proxy ricordiamo essere il client HTTP che manderà le richieste verso il relativo microservizio)
- RidesManagerService ha il solo compito di istanziare il RidesManagerController e RidesManagerImpl
- RidesManagerImpl è il cuore del microservizio → si noti che tale componente possiede delle dipendenze verso EbikesManagerRemoteAPI e UsersManagerRemoteAPI perchè deve interrogare tali microservizi prima di poter creare una nuova ride → si noti anche che tale componente crea un particolare tipo di verticle chiamato RidesExecutionVerticle, non come server HTTP, ma per la simulazione delle ride (aggiornamento posizione bici, livello batteria e credito utente) e propagazione del cambiamento tramite i due RemoteAPI delle bici e degli utenti e poi notifica tali cambiamenti anche al RideEventObserver (che di fatto è il RidesManagerImpl) affinché lo trasmetta al RidesManagerVerticle (si può dire che RidesManagerVerticle è un observer su RidesManagerImpl, che a sua volta è un observer su RidesExecutionVerticle)
- RidesManagerController ha il solo compito di istanziare il RidesManagerVerticle
- RidesManagerVerticle, come detto, è il server HTTP che gestisce le richieste e sottoscrizioni ad eventi provenienti dalle GUI, alle quali il

verticle sarà collegato mediante un WebSocket → più dettagliatamente, quando una GUI si sottoscrive, il verticle tiene aperto un web socket dedicato alla comunicazione e crea un consumer per gli eventi che riguardano le ride che ascolta sull'event bus e quando il manager avvisa il verticle del nuovo evento, esso pubblica un messaggio appropriato sull'event bus, che verrà poi ricevuto dai consumer, i quali provvederanno a propagarlo sul WebSocket → per quanto riguarda la chiusura del WebSocket, esso attende messaggi "unsubscribe" per chiudersi

- RideEventObserver è l'interfaccia degli ascoltatori (in questo RidesManagerVerticle e RidesManagerImpl)
- si noti che in questo servizio non è presente il repository perchè non ci interessava salvarci le ride passate o in corso.

2.2.4 Architettura delle GUI

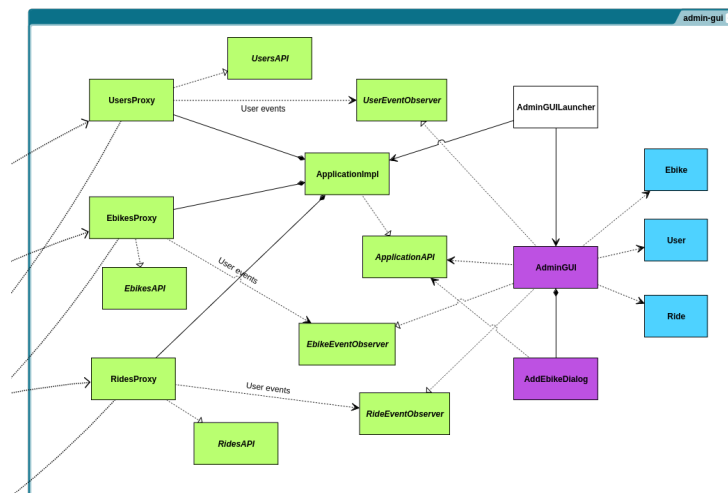


Per quanto riguarda la GUI dedicata agli utenti, essa funziona in questo modo:

- l'**UserGUILauncher** non solo lancia la GUI, rendendola di fatto visibile, ma istanzia anche **ApplicationImpl**, che racchiude tutti i metodi che utilizzano l'applicazione, raggruppati in gruppi per l'aspetto su cui concentrano (dunque metodi per utenti, bici o corse)
- **UserGUI** mostra la lista degli utenti disponibili utilizzando **ApplicationImpl**

- a seguito del login è possibile visualizzare il proprio credito e ricaricarlo, sempre attraverso ApplicationImpl (che ricordiamo contenere l'UsersProxy, in grado, attraverso l'API gateway, di comunicare all'UsersManagerVerticle, il server HTTP del servizio utenti)
- dalla medesima schermata è anche possibile avviare una nuova ride, sempre per mezzo di ApplicationImpl → in tal caso viene prima mostrata la lista di tutte le biciclette attualmente disponibili al noleggio → una volta partito il noleggio, la GUI si sottoscrive tramite l'evento "subscribe" del Proxy per gli eventi riguardanti la ride e l'utente (cambio posizione, aggiornamento credito), infatti UserGUI implementa le varie interfacce EventObserver, cosicchè i proxy delle ride e degli utenti la possano notificare.

Invece nello schema sottostante è riportata la struttura della GUI dedicata agli amministratori:



Essa funziona in modo molto simile alla GUI per gli utenti.

L'unica differenza è che, mentre la UserGUI si sottoscrive solo agli eventi generati da un unico utente e da un'unica ride, essa deve invece sottoscrivere a tutti gli eventi generati da tutti gli utenti, tutte le bici e tutte le ride.

Capitolo 3

PATTERN PER ARCHITETTURA A MICROSERVIZI

In questa sezione, vengono descritti i principali pattern utilizzati nel progetto di bike-sharing, con un focus sull'integrazione dei pattern per microservizi rilevanti al caso di studio.

Sono incluse le seguenti categorie: pattern a livello applicativo, pattern di deployment e pattern di osservabilità.

3.1 Pattern application-level

3.1.1 API Gateway

L'**API Gateway** è stato implementato come un microservizio separato, usando Spring Boot, ovvero una tecnologia differente rispetto a tutti gli altri microservizi, che infatti sfruttano Vertx.

Questo componente funge da punto di ingresso per tutte le richieste dei client, fornendo un'interfaccia unificata per accedere ai servizi sottostanti.

L'adozione dell'API Gateway comporta numerosi vantaggi:

- **incapsulamento** delle API dei microservizi → i client non accedono direttamente ai servizi, ma passano attraverso l'API Gateway, rendendo l'architettura più robusta e modulare
- **implementazione di edge functions** → funzionalità come autenticazione, autorizzazione e logging possono essere potenzialmente centralizzate in modo semplice nel gateway

- **riduzione della complessità per i client** → i client comunicano con un unico endpoint, semplificando lo sviluppo e migliorando l'esperienza utente.

Nel nostro caso, l'API Gateway utilizza **Spring Cloud Gateway** e definisce rotte per i diversi servizi, ad esempio:

```
.route("REGISTRY_ROUTE", r -> r.path("/api/registry/**")
    .uri("http://localhost:9000"))
```

3.1.2 Circuit Breaker

Nonostante non fosse richiesto tra i pattern obbligatori, per garantire resilienza e affidabilità, abbiamo implementato anche il pattern del **Circuit Breaker** (direttamente nell'API Gateway).

L'utilizzo di questo pattern consente di:

- prevenire **fallimenti a cascata** → se un servizio non risponde o è sovraccarico, il Circuit Breaker interrompe ulteriori richieste verso quel servizio
- fornire **fallback appropriati** → in caso di errore, viene restituita una risposta predefinita, garantendo una migliore esperienza utente.

Nel progetto, il Circuit Breaker è stato implementato tramite Spring Cloud Gateway, utilizzando il seguente approccio per ogni rotta:

```
.route("REGISTRY_ROUTE", r -> r.path("/api/registry/**")
    .filters(f -> f.circuitBreaker(c ->
        c.setName("registryCircuitBreaker")
        .setFallbackUri("forward:/fallback/registry"))))
```

3.1.3 Service Discovery

Il **Service Discovery** viene gestito a livello di deployment utilizzando Docker.

Nelle chiamate REST, i servizi vengono identificati tramite il nome dell'immagine del container, consentendo di localizzare e interrogare il servizio appropriato senza configurazioni aggiuntive.

Docker utilizza una **bridge network** per creare un ambiente isolato, ossia una rete virtuale, in cui i container comunicano tra loro attraverso nomi di servizio (ad esempio, `http://registry:9000`).

3.2 Pattern per deployment

Il deployment dei microservizi è stato implementato seguendo il pattern **Service-as-Container**, utilizzando Docker per incapsulare ciascun servizio in un container indipendente.

E' stato scelto tale approccio per sfruttare i seguenti benefici:

- **isolamento** → ogni container ha il proprio ambiente di runtime, garantendo l'indipendenza dei servizi
- **portabilità** → i container possono essere facilmente distribuiti su qualsiasi piattaforma compatibile con Docker
- **scalabilità** → è possibile creare più istanze di un servizio semplicemente avviando nuovi container.

Ad esempio, il deployment di un microservizio come **Registry** avviene tramite un **Dockerfile** che definisce l'ambiente e le dipendenze necessarie.

3.3 Pattern per osservabilità

In ottica di sviluppare microservizi production-ready sono stati implementati diversi pattern di osservabilità.

3.3.1 Health Check API

Il pattern **Health Check API** consente di monitorare lo stato dei servizi in produzione.

Come affermato in classe, Docker fornisce supporto nativo per i controlli di integrità (*health checks*), utilizzati per determinare se un servizio è pronto a ricevere richieste.

Nel progetto, i controlli di integrità sono configurati per ciascun servizio in modo analogo all'esempio sotto riportato per il servizio registry **Registry**, come mostrato di seguito:

```
healthcheck:
  test: curl --fail http://registry:9000/health || exit 1
  interval: 1m30s
  timeout: 10s
  retries: 3
  start_period: 40s
  start_interval: 5s
```

Più dettagliatamente:

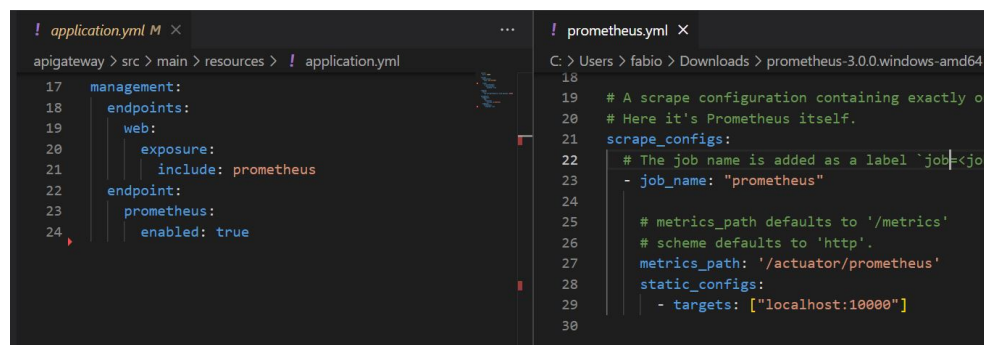
- **test** esegue un comando per verificare se il servizio è in esecuzione e risponde correttamente
- **timeout** definisce il tempo massimo per ottenere una risposta
- **retries** specifica il numero massimo di tentativi in caso di errore.

3.3.2 Application Metrics

Come affermato durante le lezioni, i microservizi mantengono delle metriche che è bene monitorare, come contatori e misurazioni varie, che vengono esposte ad un metrics server, nel nostro caso Prometheus (con Grafana per una più comoda visualizzazione grafica delle metriche tramite dashboard). Il monitoraggio di tali metriche è essenziale siccome potenzialmente permetterebbe anche di impostare soglie e avvertimenti in caso di superamenti di soglie critiche.

Nel nostro caso, per semplicità, abbiamo deciso di monitorare le metriche esposte di default da Prometheus sull'endpoint `actuator/prometheus`, così da avere una panoramica piuttosto dettagliata sullo stato delle risorse e della rete consumate dai nostri servizi.

Il servizio responsabile della raccolta ed esposizione delle metriche, come suggerito dalle best practice, è l'API gateway, il quale poi le espone a Prometheus: per fare questo passo sono state necessarie due semplici modifiche al file `application.yml` (file di Spring Boot responsabile delle configurazioni dell'API gateway) ed al file `prometheus.yml`.



The screenshot shows a code editor with two files open. The left file is `application.yml` located at `apigateway > src > main > resources > application.yml`. It contains the following configuration:

```

17 management:
18   endpoints:
19     web:
20       exposure:
21         include: prometheus
22   endpoint:
23     prometheus:
24       enabled: true

```

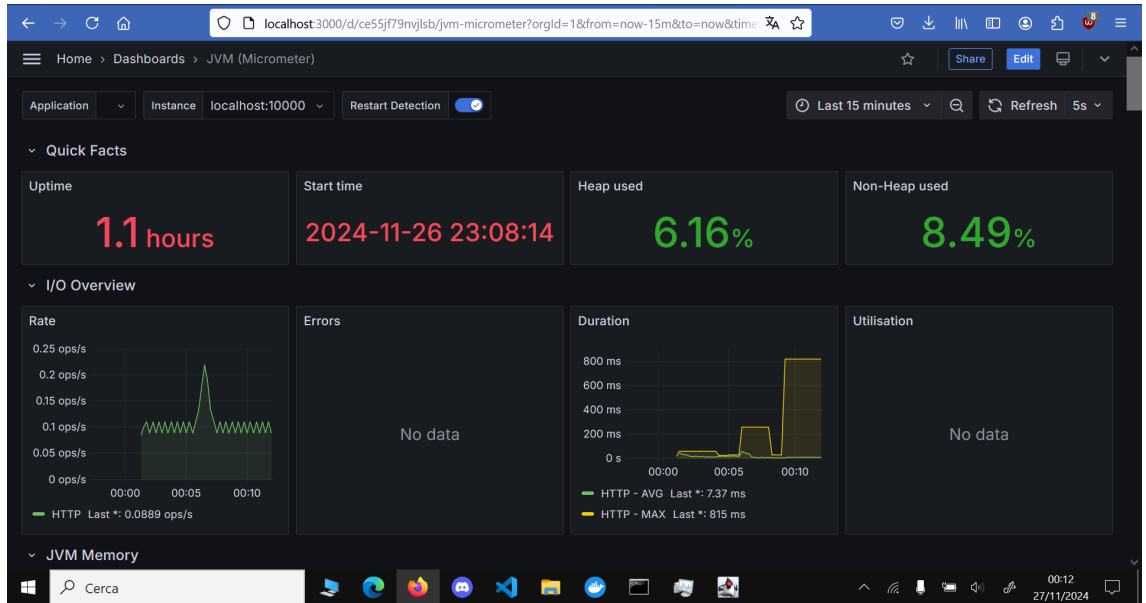
The right file is `prometheus.yml` located at `C:\Users> fabio > Downloads > prometheus-3.0.0.windows-amd64 >`. It contains the following configuration:

```

18
19 # A scrape configuration containing exactly one endpoint to scrape
20 # Here it's Prometheus itself.
21 scrape_configs:
22   # The job name is added as a label `job` to the timeseries
23   - job_name: "prometheus"
24
25     # metrics_path defaults to '/metrics'
26     # scheme defaults to 'http'.
27     metrics_path: '/actuator/prometheus'
28     static_configs:
29       - targets: ["localhost:10000"]
30

```

Così facendo e configurando poi Grafana per avere come data source proprio le metriche che provengono dall'endpoint `/actuator/prometheus` di Prometheus è possibile monitorare tutte le misurazioni in tempo reale tramite una comoda interfaccia web:



3.3.3 Pattern di configurazione a runtime

Come detto in classe, per raggiungere l'obiettivo di produrre microservizi production-ready è altresì necessario progettare i servizi in modo che siano configurabili.

Un servizio ha infatti tipicamente bisogno che gli vengano specificate varie proprietà di configurazione, fortemente dipendenti dall'ambiente che circonda il sistema a runtime.

Pertanto è importante che le proprietà di configurazione a runtime siano fornite ai servizi usando dei meccanismi di configurazione esternalizzati, nel nostro caso tramite modello push, in cui è l'infrastruttura di deployment (Docker) che passa le configurazioni ai servizi attraverso variabili d'ambiente impostate nel file docker-compose.yml:

```
docker-compose.yml
3  services:
93
94      rides:
95          build:
96              context: ./rides-manager
97              dockerfile: docker-maven/Dockerfile
98          image: rides
99          container_name: my-rides
100          depends_on:
101              - apigateway
102              - registry
103              - users
104              - ebikes
105          networks:
106              - ebikapp_network
107          ports:
108              - "9300:9300"
109          expose:
110              - 9300
111          healthcheck:
112              test: curl --fail http://rides:9300/health || exit 1
113              interval: 30s
114              timeout: 10s
115              retries: 3
116          environment:
117              - REGISTRY_URL=http://registry:9000
118              - USERS_URL=http://users:9100
119              - EBIKES_URL=http://ebikes:9200
120              - RIDES_URL=http://rides:9300
121
```

Tali variabili d'ambiente possono poi essere facilmente recuperate e lette dai singoli microservizi, in modo non solo da delineare una chiara separazione tra configurazione e codice applicativo, ma anche gestire in modo centralizzato parametri sensibili.

Capitolo 4

PATTERN NON ADOTATI

Per completezza e anche in ottica di sviluppi futuri ci siamo a lungo interrogati su come si potrebbero implementare i pattern visti a lezione ma non adottati nella nostra soluzione: ciò che è emerso in questa intensa fase di brainstorming è riportato nelle sezioni che seguono.

4.1 Pattern saga

Il pattern Saga consente una più corretta gestione delle transazioni distribuite nei microservizi.

In particolare, esso consente di coordinare una serie di operazioni che coinvolgono più servizi senza utilizzare transazioni distribuite tradizionali (che non scalano bene).

Un Saga è composto da una sequenza di transazioni locali, ciascuna con un'operazione compensativa per annullare l'operazione se necessario. Esistono due strategie principali per adottare questo pattern:

- choreography → i servizi comunicano direttamente tra loro tramite eventi
- orchestration → un coordinatore centrale coordina l'intera transazione Saga.

Il pattern Saga si rivela particolarmente utile per garantire eventual consistency e rollback distribuiti.

Qualora volessimo implementare questo pattern, potremmo sfruttare la libreria Apache Kafka.

I passi da effettuare sarebbero i seguenti:

- definizione di una Saga → per prima cosa occorre individuare dei processi che potrebbero essere implementati come transazioni distribuite, ipotizziamo ad esempio di volerlo fare per il processo di prenotazione di

una bici → come facile immaginare, questo complesso processo richiede eventual consistency ed in caso di errore in un passo occorre eseguire il rollback tramite azioni compensative

- scelta del tipo di Saga → per semplicità, ipotizziamo di scegliere un Saga a coreografia, così sfruttiamo comunicazione asincrona basata su eventi ed assenza di un orchestratore centrale
- configurazione del Saga su Kafka → con Kafka è facilmente possibile specificare gli eventi (topic) interessanti per il nostro processo (ad esempio `saga.bookRide`, `saga.paymentProcessed` o `saga.rideConfirmed`) → bisogna poi modificare ogni servizio affinché possa pubblicare/consumare questi eventi Kafka secondo necessità
- probabilmente occorrerà aggiungere qualche microservizio di supporto, in questo caso un microservizio dedicato alla gestione dei pagamenti
- i servizi dovranno poi anche prevedere delle azioni di rollback in casi critici o di fallimento (ad esempio pagamento rifiutato)
- testing, sia unit test che end-to-end, avendo cura di verificare non solo lo scenario di successo ma anche di fallimento.

4.2 Pattern event sourcing

Il pattern event sourcing si basa sul concetto che lo stato di un'applicazione non sia memorizzato come un'istantanea, ma come una sequenza di eventi che descrivono tutte le modifiche avvenute nel tempo.

Questa metodologia è utile per garantire che ogni modifica allo stato sia tracciabile e per ripristinare lo stato in caso di necessità.

Se volessimo adottare tale pattern nel nostro progetto dovremmo seguire più o meno i seguenti passaggi:

- creazione dell'event store
- introduzione della gestione degli eventi → occorre ripensare e ristrutturare l'intera applicazione affinché i microservizi siano in grado di pubblicare e consumare eventi dall'event store al verificarsi di operazioni che cambiano lo stato del sistema
- per ripristinare lo stato di un'entità (ad esempio una ride), ovvero conoscere il suo stato attuale, basta sfruttare il metodo `apply()`, che carica tutti i suoi eventi in ordine memorizzati sull'event store e li applica in ordine cronologico.

4.3 Pattern API composition

Il pattern API Composition agisce come un aggregatore che unisce le chiamate a più microservizi differenti e combina i risultati in un'unica risposta per il client.

Questo è particolarmente utile quando una singola richiesta dell'utente richiede dati provenienti da più fonti.

L'eventuale applicazione di questo pattern al nostro progetto può essere semplificato progettando l'aggregatore, ossia creando un microservizio dedicato o sfruttare l'API gateway per chiamare i microservizi interpellati ed aggregare le loro risposte.

Tale modulo può facilmente essere implementato sfruttando alcune consolidate tecnologie come RestTemplate o WebClient, che facilitano l'orchestrazione delle chiamate e la composizione della risposta finale aggregata.

4.4 Pattern CQRS

Questo pattern consente di gestire in modo più flessibile le operazioni di lettura e scrittura e permette di adattare ciascuna parte indipendentemente.

Gli eventuali passaggi per implementarlo nel nostro progetto sarebbero, per ogni servizio:

- split del database/repository in command-side e query-side
- differenziazione delle richieste in comandi e query
- testing.

4.5 Pattern di aggregazione di log

Il pattern di log aggregation centralizza la raccolta e la gestione dei log provenienti da più servizi.

Questo consente di monitorare, analizzare e visualizzare i log in tempo reale, migliorando la visibilità sullo stato dell'applicazione e facilitando la diagnosi e la risoluzione dei problemi.

L'eventuale adozione di questo pattern nel nostro progetto dovrebbe seguire l'iter seguente:

- scelta della tecnologia per la raccolta, elaborazione e visualizzazione dei log, ad esempio Elasticsearch+Kibana

- ristrutturazione dei microservizi affinché scrivano log in un formato standard (tipo JSON) e conseguente integrazione di una libreria come Logback o Log4j per inviare tali log al sistema di aggregazione
- visualizzazione e monitoraggio dei log in tempo reale tramite comode dashboard offerte da Kibana, ad esempio.

4.6 Pattern distributed tracing

Il distributed tracing è un pattern che consente di monitorare e tracciare il flusso di richieste attraverso i diversi servizi.

In esso, ogni richiesta viene tracciata tramite identificatori univoci e questo aiuta a rilevare i colli di bottiglia, diagnosticare errori e migliorare la visibilità complessiva delle performance del sistema.

In altre parole, esso permette di visualizzare il percorso di una richiesta dall'inizio alla fine, anche quando attraversa più servizi.

La sua adozione nel nostro progetto, ipotizzando di sfruttare la tecnologia Zipkin, sarebbe:

- integrazione di Zipkin nel progetto per agevolare la generazione e la propagazione dei tracciamenti delle richieste tra microservizi
- propagazione → modificare le richieste HTTP affinché includano gli header di tracing (e quindi il trace-id)
- invio dei dati al server Zipkin tramite un endpoint adeguato e dedicato
- visualizzazione e monitoraggio tramite interfaccia web di Zipkin.

4.7 Pattern exception tracking

Il pattern exception tracking si occupa di catturare, registrare e gestire le eccezioni per monitorare e risolvere i problemi.

Consiste in pratica nell'implementare una strategia centralizzata che permetta di identificare, tracciare e rispondere in modo appropriato agli errori che si verificano.

Per adottare tale pattern nel nostro progetto dovremmo:

- centralizzare la gestione delle eccezioni creando una classe dedicata (ciò è facilitato dal tag `@ControllerAdvice` di Spring Boot, che gestisce le eccezioni a livello di controller)
- fare logging delle eccezioni accadute e successivo monitoraggio, magari integrando strumenti come Prometheus

- personalizzare le risposte in modo da restituire messaggi significativi e comprensibili agli utenti e mantenere la sicurezza (evitando di esporre dettagli interni), ad esempio creando risposte HTTP personalizzate per errori specifici.

4.8 Pattern audit logging

Il pattern di audit logging riguarda la registrazione sistematica di eventi rilevanti all'interno di un'applicazione per garantire trasparenza e sicurezza. Più dettagliatamente, prevede di monitorare chi ha eseguito una certa azione rilevante, cosa è stato fatto, quando è successo e dove è accaduto.

Volendo implementare tale pattern all'interno del nostro progetto dovremmo seguire i seguenti passi:

- identificazione dei punti di audit, ossia delle azioni che meritano di essere registrate, in quanto sensibili o rischiose
- definizione di un servizio dedicato all'audit, che riceve gli eventi rilevanti dagli altri microservizi
- implementazione di log rilevanti ed esaurienti, in grado di individuare con precisione chi ha fatto un'azione, in cosa consiste l'azione e quando
- ristrutturazione dei microservizi affinché inviino gli eventi accaduti al servizio dedicato all'audit
- progettazione di strategie per preservare i log, ad esempio salvandoli su database
- progettazione di strumenti comodi ed efficaci per accedere ai log, ad esempio con Kibana o Grafana.

Capitolo 5

TESTING

Come richiesto nella consegna, è stato inserito un esempio di test per ogni livello della Test Pyramid.

5.1 Unit test

E' stato inserito un unit test (file `ebikes-manager/test/java/sap/EbikeTest.java`) per testare che la classe `Ebike.java` lavorasse correttamente e come atteso. Tale test, come visto a lezione, è riconducibile alla tipologia sociabile unit test, in quanto assieme alla classe sotto test sono state testate anche tutte le sue dipendenze reali, non tramite mockup come avviene nei solitary unit test.

5.2 Integration test

Un integration test verifica che un servizio interagisca in modo appropriato con altri servizi e/o con l'infrastruttura esterna.

Nel nostro test (file `ebikes-manager/test/java/sap/EbikeManagerIntegrationTest.java`), abbiamo scelto di testare la correttezza del comportamento dell'`EbikeManager`, che gestisce le comunicazioni tra il layer di dominio del microservizio delle biciclette ed il suo database, ovvero `EbikesManager`.

In particolare il nostro test sfrutta un approccio orientato ai contratti, ovvero alla verifica che la forma delle API fornite dal provider (il repository) soddisfi le aspettative del consumer (`EbikesManager`).

Si noti che questo particolare test sfrutta un mock, ovvero una copia simulata del repository, molto utile per isolare il test dal database reale, ovvero assicurarsi che il test non fallisca per problemi terzi legati al database vero e proprio o a fattori esterni.

5.3 Component test

Il component testing verifica la correttezza del comportamento di un servizio in isolamento.

Nel nostro caso il component test è stato realizzato per verificare il corretto funzionamento del microservizio Ebikes, sfruttando il framework Cucumber per la definizione dei test in Gherkin e strumenti come Testcontainers e RestAssured per il deployment del microservizio in un ambiente isolato e per l'invio delle richieste HTTP.

Più nel dettaglio:

- Gherkin è utilizzato per definire scenari di test leggibili e orientati al dominio, come interazioni tramite API HTTP (GET/POST)
- Cucumber traduce gli step definiti in Gherkin in codice eseguibile, permettendo l'integrazione vera e propria
- RestAssured è una libreria Java utilizzata per inviare richieste e verificare le risposte del microservizio
- Testcontainers avvia e gestisce il container docker delle bici, assicurando isolamento e riproducibilità del test.

Il test è composto da tre passi principali:

- creazione del file `ebikes.feature`, dove sono descritti in linguaggio Gherkin gli scenari da verificare, in un modo leggibile e domain-oriented
- definizioni degli step nel file `EbikesStepDefinitions` → tali step implementano le azioni e verifiche richieste dagli scenari Gherkin, ossia per ogni scenario individuato si fa il setup del microservizio, si inviano le richieste, si verificano i risultati restituiti e infine si arresta il container Docker per liberare le risorse
- esecuzione del test tramite il runner `EbikeComponentTest`, ossia una semplice classe JUnit a cui occorre solamente specificare il percorso del feature file e gli step definiti.

5.4 End-to-end Test

Il testing end-to-end (E2E) ha l'obiettivo di verificare il funzionamento dell'intera applicazione, simulando un flusso completo che coinvolga tutti i suoi componenti principali.

Questo tipo di test è particolarmente utile per individuare eventuali problemi di integrazione tra i servizi, garantendo che l'applicazione funzioni correttamente come sistema unitario.

5.4.1 Strategia User Journey

Per testare la nostra applicazione, abbiamo adottato la strategia *user journey*: questa consiste nella simulazione di un flusso d'uso tipico, eseguendo un unico test che racchiude i principali casi d'uso del sistema.

In particolare, il test include le seguenti operazioni:

- creazione di un utente
- ricarica del credito per l'utente
- recupero degli ID delle biciclette disponibili
- noleggio di una bicicletta
- restituzione della bicicletta.

Il test è stato implementato utilizzando la libreria *RestAssured*, che consente di effettuare chiamate HTTP e verificare le risposte.

Il flusso completo è descritto nel codice seguente:

```
public class E2ETest {
    @BeforeAll
    public static void setup() {
        String apiGatewayUrl = System.getenv("APIGATEWAY_URL");
        RestAssured.baseURI = apiGatewayUrl;
    }

    @Test
    public void testUserJourney() {
        // Passaggi principali del flusso
    }
}
```

5.4.2 Esecuzione con Docker Compose

Per eseguire il test end-to-end, abbiamo utilizzato *Docker Compose*, che permette di orchestrare i vari servizi dell'applicazione in un ambiente isolato. È stato pertanto creato un nuovo container dedicato al test, configurato nel file `docker-compose.yml` con il seguente servizio:

```
testrunner:
  build:
    context: ./end2end-test
    dockerfile: docker-maven/Dockerfile
  image: testrunner
```

```
container_name: my-test-runner
depends_on:
  - apigateway
  - registry
  - users
  - ebikes
  - rides
networks:
  - ebikapp_network
```

Il servizio `testrunner` dipende dai container principali (`apigateway`, `registry`, `users`, `ebikes`, `rides`) e li utilizza per eseguire il test: ciò garantisce che il test end-to-end venga eseguito in un ambiente realistico, simulando l'infrastruttura completa.

Capitolo 6

TESTING DEI QUALITY ATTRIBUTE

Durante la fase di analisi avevamo dedicato del tempo ad individuare i seguenti quality attribute ritenuti rilevanti:

- tutti i medesimi quality attribute individuati nell'assignment precedente
- modularità
- performance degli endpoint
- disponibilità.

Nelle sezioni seguenti si descrivono i quality attribute scenarios dei quality attribute che sono stati espressamente verificati.

6.1 Modularità

La modularità misura il grado di separazione e indipendenza dei componenti di un sistema.

Un sistema modulare consente modifiche locali a un componente senza impatti significativi sugli altri.

Nel contesto della nostra architettura a microservizi, ciò implica che ogni servizio opera in autonomia, con interazioni definite tramite API ben strutturate.

Quality attribute scenario:

- stimolo → viene richiesta una modifica/aggiornamento ad un micro-servizio
- sorgente dello stimolo → stakeholder

- artefatto → servizio interessato e sua struttura di deployment
- ambiente → durante il normale ciclo di sviluppo, con l'intero sistema distribuito in funzione
- risposta → la modifica viene implementata nel microservizio interessato e nella sua struttura di deployment, senza alterare gli altri servizi
- misura dalla risposta → il servizio aggiornato viene distribuito e riavviato entro pochi minuti senza downtime per gli altri container.

6.2 Performance degli endpoint

La performance degli endpoint misura la velocità con cui un servizio risponde a una singola richiesta.

Nel contesto della nostra architettura a microservizi, ciò implica che ogni endpoint deve fornire risposte rapide e consistenti sotto condizioni normali.

Quality attribute scenario:

- stimolo → una singola richiesta viene inviata ad un endpoint specifico
- sorgente dello stimolo → un utente del sistema
- artefatto → endpoint del microservizio
- ambiente → durante il normale funzionamento del sistema distribuito
- risposta → l'endpoint elabora e restituisce la risposta entro un tempo definito
- misura dalla risposta → il tempo di risposta è inferiore a 200 ms.

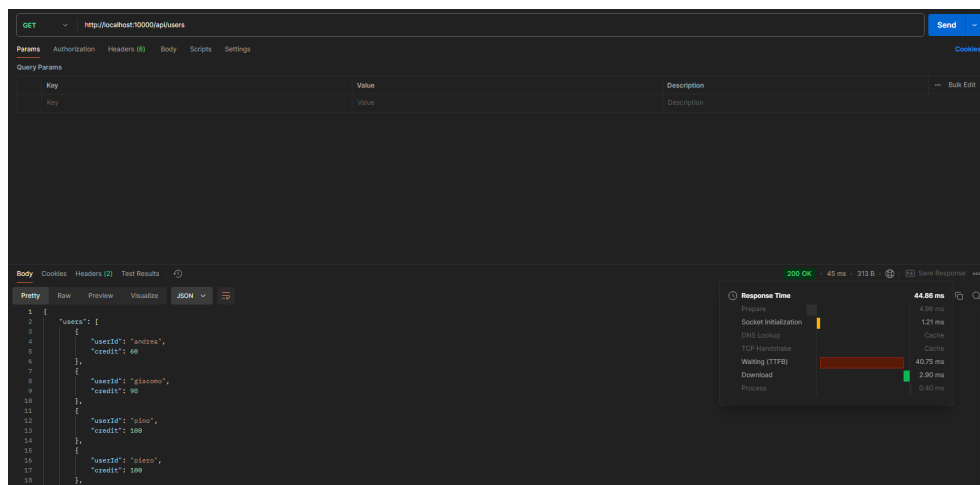


Figura 6.1: come si nota, la richiesta viene servita in molto meno di 200 ms.

6.3 Disponibilità

La disponibilità misura la capacità del sistema di essere operativo e rispondere alle richieste, includendo la corretta orchestrazione e avvio dei microservizi.

Nel contesto della nostra architettura, ciò implica che un microservizio possa essere avviato rapidamente e che l'orchestrazione gestisca correttamente la sua integrazione.

Quality attribute scenario:

- stimolo → un microservizio viene riavviato per manutenzione o a seguito di un guasto
- sorgente dello stimolo → infrastruttura di deployment o team di sviluppo
- artefatto → microservizio e sistema di orchestrazione
- ambiente → durante il normale funzionamento del sistema distribuito
- risposta → il microservizio viene avviato correttamente e l'orchestrazione ne garantisce l'integrazione con gli altri servizi
- misura dalla risposta → il microservizio è completamente operativo entro 30 secondi senza interruzioni per gli altri servizi.


```

2024-11-30 17:17:29 Nov 30, 2024 4:17:29 PM sap.ass2.users.infrastructure.UsersManagerVerticle getAllUsers
2024-11-30 17:17:29 INFO: Received 'getAllUsers'
2024-11-30 17:17:36 [INFO] Scanning for projects...
2024-11-30 17:17:37 [INFO]
2024-11-30 17:17:37 [INFO] -----< sap.ass2:user-manager >-----
2024-11-30 17:17:37 [INFO] Building user-manager 1.0-SNAPSHOT
2024-11-30 17:17:37 [INFO]   from pom.xml
2024-11-30 17:17:37 [INFO] -----[ jar ]-----
2024-11-30 17:17:38 [INFO]
2024-11-30 17:17:38 [INFO] --- exec:3.5.0:java (default-cli) @ user-manager ---
2024-11-30 17:17:41 Nov 30, 2024 4:17:41 PM sap.ass2.users.infrastructure.UsersManagerVerticle getAllUsers
2024-11-30 17:17:41 INFO: Received 'getAllUsers'

```

Figura 6.2: si noti che alle 17:17:29 è stata servita una richiesta, poi il microservizio è stato riavviato per un problema tecnico ma già alle 17:17:41 era nuovamente operativo e ha servito una nuova richiesta.