UNIVERSITÀ
DEGLI STUDI
FIRENZE

StreetApp

Implementazione di un architettura a microservizi orientata agli eventi per la gestione di graffiti.

Alessandro Mini
7060381

Febbraio 2022

UNIVERSITÀ DEGLI STUDI FIRENZE

Indice

UNIVERSITÀ DEGLI STUDI FIRENZE

StreetApp

In questo progetto si implementa il **back-end** per l'applicazione StreetApp, un app. social per la gestione di Graffiti nelle città.

Un graffito è identificato come un *artwork*, un artefatto caratterizzato da:

1. Una risorsa (.JPG, .GIF ..).
2. Coordinate espresse come (*lat*, *long*).
3. L'autore dell'artwork (può non essere noto)
4. Movimento artistico/crew di riferimento.
5. Tipo di arte (pittura, murales ...).
6. Stile artistico (comic art, iper-realismo ...)

StreetApp

Gli **obiettivi** di questo progetto sono:

- Implementare un sistema back-end ibrido per StreetApp che combini l'approccio classico SOA con quello event driven.
- Studiare procedimenti di design per identificare e progettare microservizi.
- Implementare il tutto secondo il paradigma architetturale REST.
- Studiare pattern specifici e elementi di architetture di questo tipo.
- Studiare ed implementare un opportuna suite di test.

I pattern centrali analizzati nell'implementazione sono:

- **Event Sourcing :** per orientare ad eventi la gestione degli artwork, garantendone quindi possibilità di storicizzazione.
- **CQRS :** per la gestione disaccoppiata di commands e queries.
- **Aggregate:** per centralizzare i comandi e gestire gli eventi.

UNIVERSITÀ DEGLI STUDI FIRENZE

StreetApp

Il lavoro fatto ha previsto i seguenti passi:

1. Studiare i requirements del sistema.
2. Studiare il domain model, identificare i *bounded contexts*, i microservizi e gli eventi.
3. Scegliere le tecnologie per lo sviluppo.
4. Analizzare i pattern sopracitati (CQRS, EventSourcing, Aggregate) e la loro interazione.
5. Costruire un implementazione in Java (con relative basi di dati) dei 3 servizi identificati al punto (1).
6. Costruire una suite di test adeguata.
7. Studiare un caso di uso reale.
8. Eseguire un analisi critica delle architetture utilizzate.

UNIVERSITÀ DEGLI STUDI FIRENZE

Analisi dei Requirements

Un primo step che guiderà tutto il processo di sviluppo è andare a descrivere i requirements funzionali e non funzionali.

- **Requirements funzionali:**

1. Creazione di un utente (un utente si può registrare sull'applicazione).
2. Creazione di un Autore (gli autori potrebbero registrarsi direttamente o "essere registrati" dagli utenti).
3. Un artwork è segnalato da un utente.
4. Modifica di un artwork di tipo "standard" es. viene modificato il nome, o alcune sue caratteristiche.
5. Modifica di un artwork "drastico": un artwork viene sovrascritto (cambia cioè in tutte le sue caratteristiche tranne latitudine e longitudine).
6. Possibilità di eseguire queries di vario tipo sui servizi.
7. Creazione "parziale" di un artwork: Un utente segnala un artwork senza conoscere alcuni campi (Es. autore, stile, tipo ...) che potranno essere aggiunti in seguito.
8. Possibilità di interazione "storica".

Analisi dei Requirements

- **Requirements non funzionali:** I requisiti non-funzionali principali identificati sono:
  1. Gli artwork sono identificati univocamente da un ID, latitudine e longitudine non potranno essere cambiati dopo l'inserimento di un artwork.
  2. Non è presente il sistema di gestione delle risorse .jpg,.png.
  3. Gli users e gli autori sono identificati dal loro id, alcuni campi non possono essere duplicati (es. username).
  4. Un utente ha accesso *completo* al sistema, non è presente autenticazione, autorizzazione e accounting.
  5. I servizi devono essere robusti di fronte a richieste errate/malformate (con opportuna gestione delle eccezioni).
  6. I servizi devono esporre interfacce ben identificabili e aperte all'eterogeneità di tecnologie (es. diversi DBMS).

Analisi del Domain Model - Microservizi

L'obiettivo è costruire un **architettura orientata a microservizi**:
Un paradigma che prevede di implementare un sistema complesso come un *set* di servizi (distribuiti) più semplici [18].

Principi fondamentali delle architetture a microservizi

Le architetture di questo tipo dovrebbero rispettare i seguenti principi:

- **Minima responsabilità**: ogni microservizio ha la responsabilità ristretta ad una parte del domain model.
- **Loose Coupling**: i microservizi devono essere *debolmente accoppiati*.
- **Shared Libraries:** Non si devono utilizzare librerie condivise se non per elementi immutabili.
- **Divisione delle funzionalità:** ogni microservizio dovrebbe avere la propria funzionalità (completa) riferita ad una parte del D.M, es. ogni microservizio ha la propria base di dati.

Analisi del Domain Model - Microservizi

Identificare i microservizi è un task complesso, per cui sono stati proposti più approcci.

- In generale identificare i microservizi corrisponde ad un task di *decomposizione* ”logica” del sistema in servizi, in un ottica che è più vicina alla business logic che agli aspetti tecnici.

Una delle proposte è quella di utilizzare l’approccio di definizione del dominio di tipo DDD, il cui funzionamento può essere schematizzato come segue [18]:

1. Si identificano i **sottodomini** del problema in analisi, partendo dalla descrizione dell’architettura.

   L’identificazione dei sottodomini corrisponde all’identificazione delle *capabilities* dell’architettura.

2. Per ogni sottodominio individuato si costruisce un **domain model**.

3. Si definiscono le relazioni tra i sottodomini.

Si definisce un **Bounded Context** come lo scope di un domain model [18].

Analisi del Domain Model

Un importante proprietà descritta da Richardson è la seguente:

Proprietà

Si osserva sperimentalmente che ogni *bounded context* corrisponde nell'implementazione ad uno o più microservizi.
DDD e l'architettura orientata a microservizi sono quindi in "perfect alignment" [18].

Si sono svolti i seguenti **passaggi**:

1. Si sono identificati i 3 sottodomini.
2. Si è costruito un domain model per ognuno di questi.
3. Si sono definite le relazioni tra i 3 domain model identificati.
4. Si sono identificati i bounded contexts, come scope dei 3 domain models, questi sono: Artist Context, User Context, Artwork Context.

Analisi del Domain Model

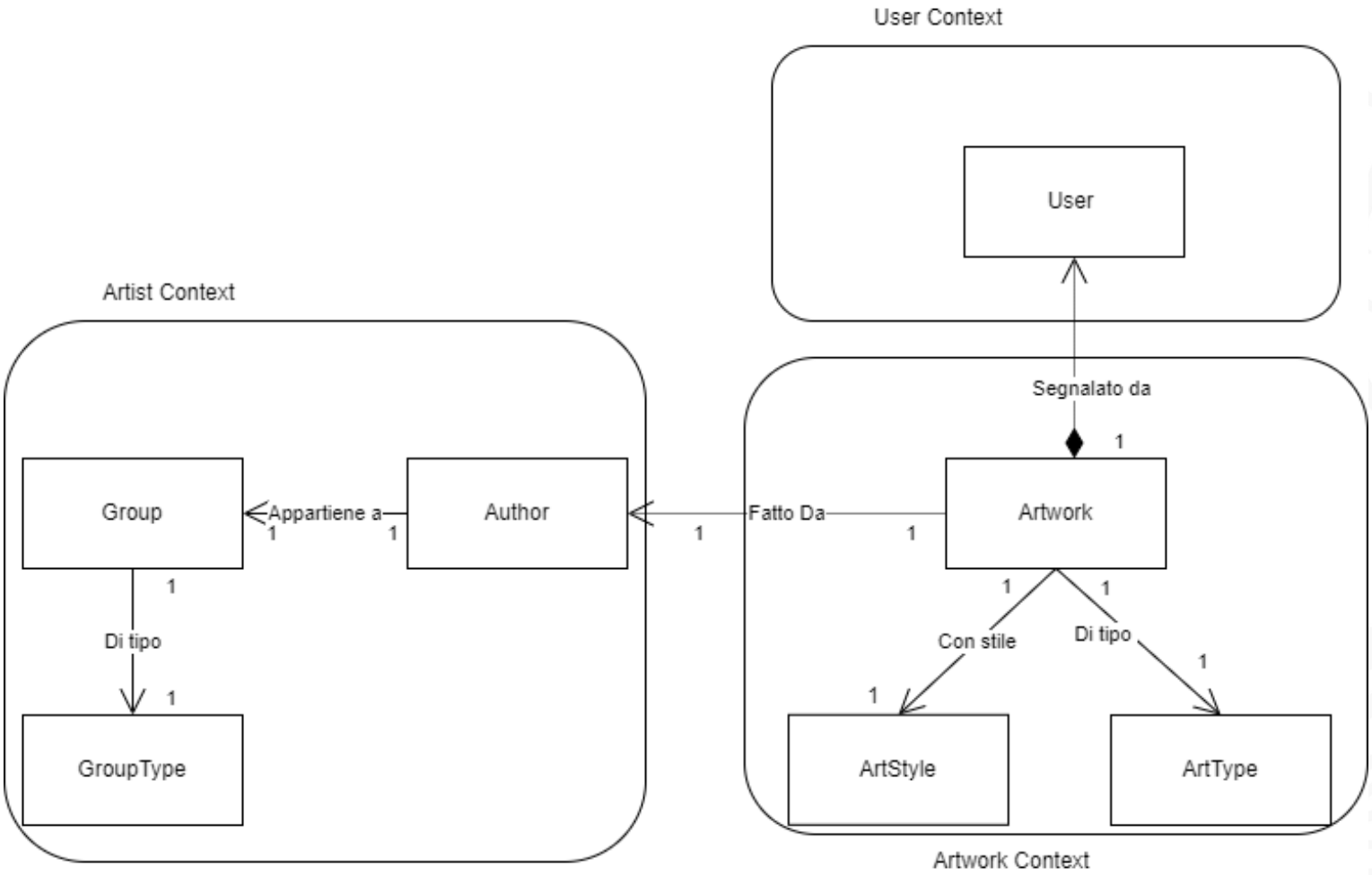


Figure 1: Domain model ”globale” con relativi bounded contexts per Street App.

UNIVERSITÀ DEGLI STUDI FIRENZE

Analisi del Domain Model - Eventi

Dai 3 bounded contexts si identificano 3 microservizi: *UserMicroservice, AuthorMicroservice, ArtworkMicroservice.*
Segue un ulteriore passo in cui si identificano gli **eventi**.

Dal momento che la parte dell'architettura agli eventi sarà ArtworkMicroservice, ci si limita all'analisi degli eventi che quest'ultimo può generare:

- *ArtworkCreatedEvent* : Un artwork viene riportato da un utente.
- *ArtworkAuthorChangedEvent* : Evento di modifica dell'autore di un artwork.
- *ArtworkNameChangedEvent* : Evento di modifica del nome di un artwork.
- *ArtworkRemovedEvent* : L'artwork viene rimosso.
- *ArtworkStyleChangedEvent* : Evento di modifica dello stile di un artwork.
- *ArtworkTypeChangedEvent* : Evento di modifica del tipo di un artwork.

Tecnologie - Patterns (CQRS)

Definizione:

CQRS è un pattern che prevede il *disaccoppiamento* tra quelli che sono i comandi (CUD) e le queries (R).

Ogni servizio implementerà dei **CommandDTO*, ad esempio *UserUsernameUpdateDTO* e delle specifiche richieste di lettura, tutto il flusso delle informazioni sarà disaccoppiato.

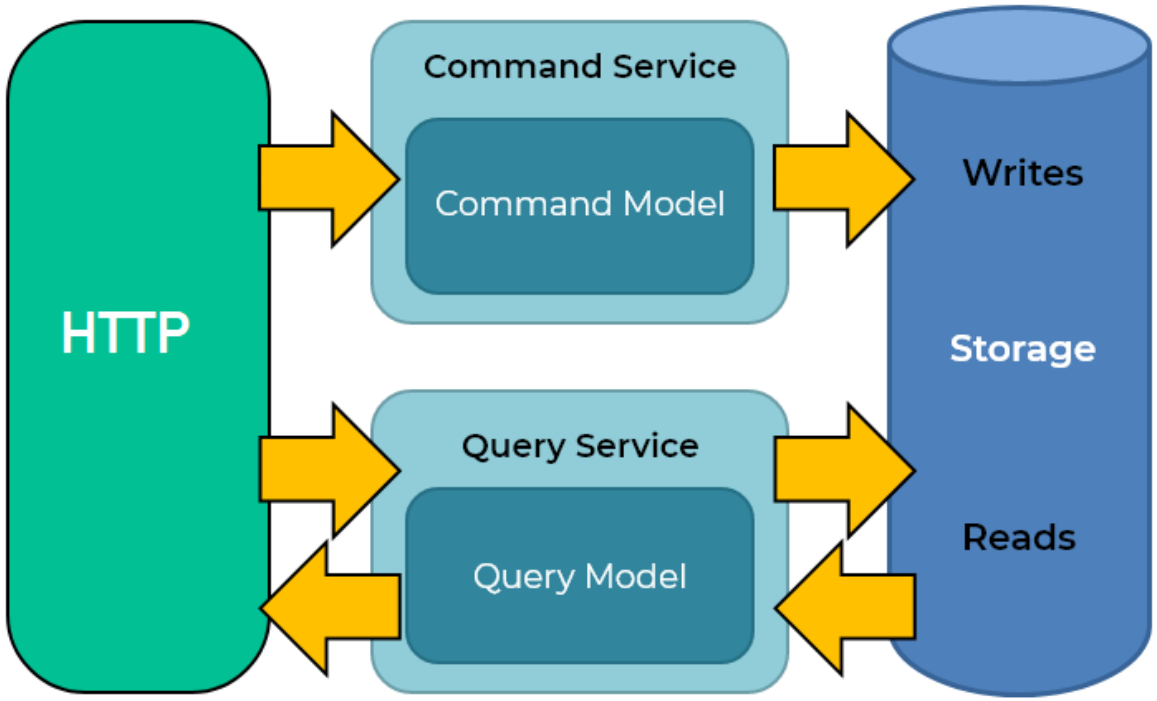
Spesso **CQRS** viene implementato con 2 basi di dati:

1. Base di dati per le letture.
2. Base di dati per le scritture.

In questo progetto si implementerà una versione in cui si ha **un solo database** (comune a letture e scritture), questo perchè:

1. Il dominio di lavoro è semplice e la presenza di due database introdurrebbe ridondanza e complessità in più, senza contare la presenza di problemi di consistenza interni ai servizi che dovrebbero essere gestiti.
2. Le funzioni di proiezione a causa della semplicità del dominio degenererebbero in funzioni identità.
3. La presenza di un unica base di dati è considerata accettabile [22], diviene consigliata nel caso in cui si utilizzi un event store in memory [5].

Patterns (CQRS)



UNIVERSITÀ DEGLI STUDI FIRENZE

Patterns (Event Sourcing)

Event Sourcing è un pattern che prevede di:

1. Gestire un entità attraverso i suoi eventi, ogni modifica genera un evento (inclusa la sua creazione.
2. Un entità è persistita come una sequenza (ordinata) di eventi.
3. Un entità viene ”ricostruita” a partire dai suoi eventi (Si crea una *nuova* entità, e vi si applicano tutti gli eventi della vecchia).

L’applicazione di questo pattern è estrememante efficente per la gestione della *storicizzazione* delle entità, in quanto queste saranno fondamentalmente una ”lista di eventi”.
In **StreetApp** la gestione degli artwork avverrà attraverso il pattern EventSourcing (un Artwork sarà una lista di eventi), questo pattern ne gestirà quindi l’evoluzione e la storicizzazione.

Aggregate

Aggregate è un pattern che permette di gestire un cluster di oggetti come un unica entità, attraverso una *root.*

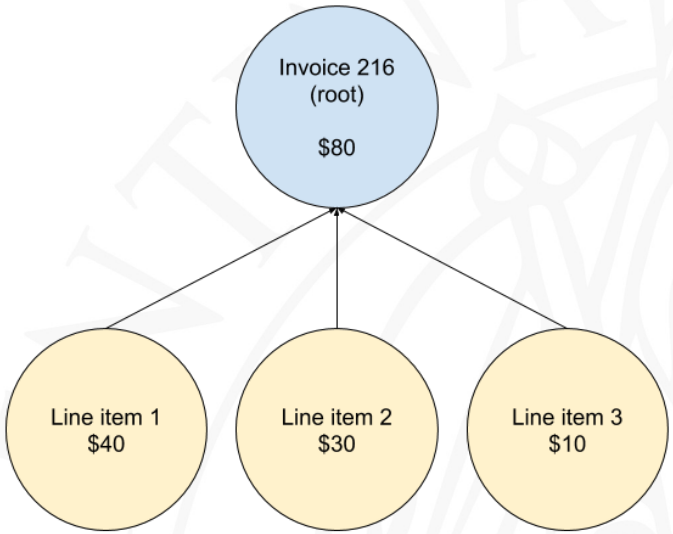


Figure 2: Visualizzazione del pattern aggregate relativo alle fatture

Questo pattern fornisce una **centralizzazione** dei comandi sul cluster di oggetti, riducendo la superficie di rischio di fallimento e migliorando l'organizzazione logica dell'architettura.

UNIVERSITÀ DEGLI STUDI FIRENZE

CQRS + ES + Aggregate

Aggregate si rivela essere *fondamentale* per l'organizzazione logica e la gestione degli eventi, in associazione con EventSourcing.

Un idea per ottenere una **sinergia** tra questi pattern è la seguente:

1. Gli artwork saranno gestiti con il pattern Aggregate, saranno un oggetto concreto con radice (root) astratta.
2. Gli artwork saranno gestiti attraverso event-sourcing, memorizzati in un eventstore.
3. Le richieste saranno disaccoppiate, i commands *aggiungeranno* eventi, le queries *ricostruiranno* lo stato di un aggregato dall'event-store.

I servizi comunicheranno attraverso i principi dello stile architetturale REST, alcune operazioni coinvolgono chiamate ad altri servizi.

UNIVERSITÀ DEGLI STUDI FIRENZE

Tecnologie - Programmazione

L'implementazione del progetto consiste in 3 progetti eclipse (Java EE), uno per microservizio.
Ho utilizzato le seguenti tecnologie per la fase di **sviluppo**:

- WildFly versione 25.0.0.Final.
- RestEASY versione 4.7.2.Final.
- javax-api versione 8.0.1.
- JBOSS Tools versione 4.21.0 Final.
- Hibernate versione 5.4.13.Final con connettore MySQL versione 8.0.27 .

Per il **testing** ho utilizzato:

- Junit versione 4.12.
- Mockito versione 4.2.0.

Ogni microservizio avrà un suo sistema di basi di dati e memorizzazione.

UNIVERSITÀ DEGLI STUDI FIRENZE

Implementazione

Viene quindi presentata l'implementazione, le slides che seguiranno saranno organizzate come segue:

1. Descrizione del modello di comunicazione e di interazione con l'utente.
2. Descrizione nel dettaglio di *UserMicroservice*, con dettagli implementativi relativi al progetto e al pattern CQRS, che verrà poi adottato negli altri 2 servizi.
3. Descrizione generale di *AuthorMicroservice* (si ometteranno i dettagli per brevità essendo la sua struttura simile ad UserMicroservice).
4. Descrizione nel dettaglio di *ArtworkMicroservice* con dettagli relativi all'adozione di Aggregate ed EventSourcing.

Seguirà poi l'analisi di funzionamento e il testing.

UNIVERSITÀ DEGLI STUDI FIRENZE

Gestione delle Eccezioni

Per la gestione delle **eccezioni** si è utilizzato il sistema delle exception user-defined e degli *ExceptionMappers*, in accordo alla documentazione di RESTEASY e Jax-RS.

Si implementano quindi delle eccezioni che modellano due casi:

1. La risorsa non è stata trovata (Resource not found exception).
2. La richiesta è malformata (BadRequest exception).

Per ogni eccezione si implementa un apposito **mapper**.

UNIVERSITÀ DEGLI STUDI FIRENZE

Interazione con l'utente finale

I 3 progetti espongono 3 servizi che interagiscono attraverso chiamate HTTP, ogni servizio ha un indirizzo del tipo :

http://localhost:8080/ServiceName/rest/ServiceName/comando

in cui ServiceName è il nome del servizio, comando è invece la caratterizzazione dell'endpoint, che risponderà secondo vari metodi HTTP.

Interazione con l'utente finale

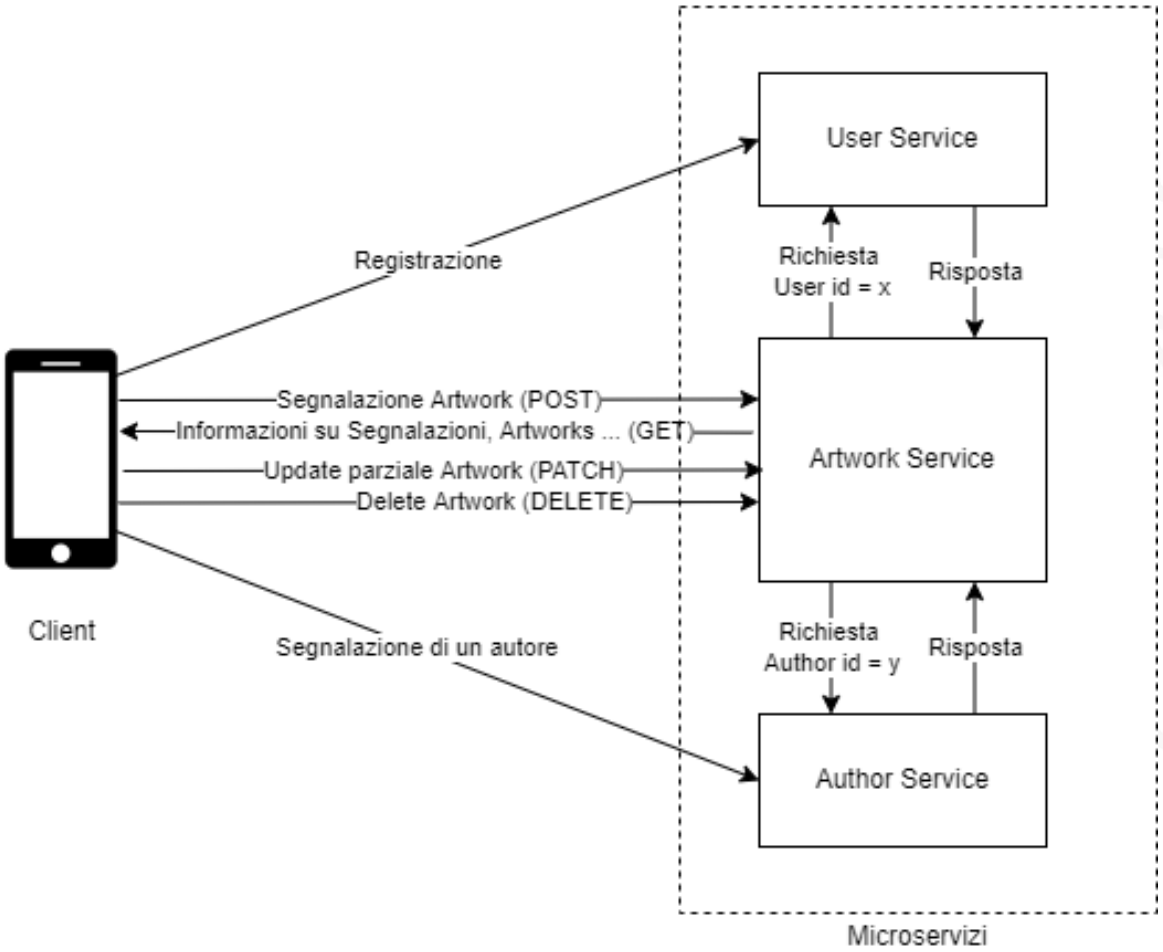


Figure 3: Comunicazione tra microservizi e utente finale.

UserMicroservice

UserMicroservice gestisce gli users, un **user** è caratterizzato dai seguenti campi:

```
private int id;
private String username;
```

Questo servizio espone i seguenti metodi:

- *POST http://.../UserMicroservice/rest/UserMicroservice*
- *PATCH http://.../UserMicroservice/rest/UserMicroservice/updateusername/{id}:*
- *DELETE http://.../UserMicroservice/rest/UserMicroservice/{id}*
- *GET http://.../UserMicroservice/rest/UserMicroservice:*
- *GET http://.../UserMicroservice/rest/UserMicroservice/{id}:*

Il programma gestirà le opportune eccezioni, segue un analisi di tutti i componenti

UNIVERSITÀ DEGLI STUDI FIRENZE

Elementi di UserMicroservice

- **endpoints**: Sono 2, impementati per rispecchiare l'approccio CQRS, *UserCommandEndpoint* che riceve comandi CUD da interfaccia HTTP e *UserViewEndpoint* che risponde con DTOs alle queries (richieste R da HTTP).
- **DAOs :** si ha un package con la classe UserDAO la quale ha i metodi create, findall, update, delete per interagire con il database attraverso JPA (Hibernate).
- **DTOs**: Si ha un package che contiene il DTO relativo all'user e alle richieste, ad esempio una richiesta di creazione di un utente (che viene tradotta in una classe Java):

```
//Richiesta di creazione di un utente
    public class UserCreationCommandDTO {
    String username;
    ...
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Dettaglio sulle richieste

Per la gestione delle richieste:

- I ***Commands*** (CQRS) sono caratterizzati da richieste, ad esempio di creazione, update o cancellazione.
- Ogni richiesta è una classe Java che veicola tutti i campi necessari alla richiesta.
- Un *endpoint* accetta un oggetto di tipo richiesta.

Questa scelta permette di:

1. Avere una *granularità* più fine nelle operazioni ammissibili.
2. Gestire con maggior accuratezza e semplicità gli eventi e la storicizzazione in ArtworkMicroservice (EventSourcing)
3. Permettere agevolmente richieste parziali, esempio nel caso in cui si debba aggiornare l'username di un utente, si invierà una richiesta PATCH di tipo *UserUsernameUpdateRequest* con il valore *"username":"newUsername"*.

UNIVERSITÀ DEGLI STUDI FIRENZE

Elementi di UserMicroservice

- **entities**: Package che contiene la classe User, con annotazione JPA, l'id di un user è gestito da JPA con la seguente metodologia di gestione:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
```

  Si impone anche che l' User abbia username unico, in modo tale da evitare User con username duplicato, come nella maggioranza delle applicazioni reali:

```
@Table(uniqueConstraints =
{ @UniqueConstraint(columnNames = { "username" }) })
```

- **Mappers**: Package che contiene il mapper per convertire l'entità User in un UserDTO e viceversa, questo package contiene anche il mapper per le eccezioni, che sarà analizzato in seguito.

UNIVERSITÀ DEGLI STUDI FIRENZE

Elementi di UserMicroservice

- **controllers**: Package che implementa i controller, un endpoint gestisce le chiamate da interfaccia HTTP ed invoca il relativo controller, i controller sono 2, *UserCommandController* e *UserViewController*.
- **exceptions** : Package che contiene eccezioni user-defined, il meccanismo di gestione delle eccezioni prevede la presenza di classi di tipo **Exception* e relativi mappers (ExceptionMappers jax-rs).
  - Ad esempio un eccezione di tipo *"ResourceNotFoundException"* rappresenta il caso in cui una certa risorsa non sia trovata nel sistema, e viene mappata in un errore 404.
  - Ad ogni eccezione si associa un relativo mapper (nel package mapper).

UNIVERSITÀ DEGLI STUDI FIRENZE

UserMicroservice - GET

Nel caso in cui l'utente esegue una richiesta **GET**:

1. La richiesta viene gestita dall'endpoint (JAX-RS) e viene inoltrata al controller (UserViewController), in cui si ha il metodo handler.
2. Il controller interagisce con il DAO.
3. il DAO interagisce con il sistema di persistenza, risponde al controller.
4. Il controller restituisce il focus all'endpoint.
5. L'endpoint restituisce il valore all'utente (con contratto JSON) o in caso di errore restituisce un eccezione:
    1. BadRequestException: la richiesta è malformata.
    2. ResourceNotFoundException: la richiesta non è stata trovata nel sistema di persistenza.

UNIVERSITÀ DEGLI STUDI FIRENZE

UserMicroservice - POST

Nel caso in cui l'utente esegue una richiesta **POST** (creazione):

1. La richiesta viene gestita dall'endpoint (JAX-RS) e viene inoltrata al controller (UserCommandController), in cui si ha il metodo handler.
2. Il controller interagisce con il DAO.
3. il DAO interagisce con il sistema di persistenza,crea l'oggetto Utente, verifica la corretta creazione e risponde al controller con l'entità creata.
4. Il controller restituisce il focus all'endpoint, passandogli l'entità creata.
5. L'endpoint:
    1. Se l'utente è stato creato risponde con un UserDTO dell'utente creato.
    2. Altrimenti restituisce un eccezione.

UserMicroservice - PATCH

Nel caso in cui l'utente esegue una richiesta **PATCH** (update parziale, es. username modificato):

1. La richiesta viene gestita dall'endpoint (JAX-RS) e viene inoltrata al controller, secondo la specifica richiesta, ad esempio

$$\text{Aggiornare username} -> \text{UserUsernameUpdateRequest}$$

2. Il controller interagisce con il DAO.
3. il DAO interagisce con il sistema di persistenza, aggiorna l'username sul database, e risponde al controller con il riferimento all'entità aggiornata.
4. Il controller restituisce il focus all'endpoint, passandogli il riferimento all'entità aggiornata.
5. L'endpoint risponderà all'user, o restituirà un eccezione.

Si omette la descrizione dell'operazione **DELETE**.

UNIVERSITÀ DEGLI STUDI FIRENZE

AuthorMicroservice

AuthorMicroservice ha un funzionamento analogo a UserMicroservice, quindi verrà omesso nella sua interezza.
Un **autore** è caratterizzato dai seguenti campi:

```
private int id;
private String authorName;
private String groupName;
private groupMovement groupType;
```

La struttura è uguale a quella di UserMicroservice con lo stesso tipo di chiamate HTTP e meccanismo di persistenza, la trattazione procederà ad analizzare ArtworkMicroservice:

1. Il focus sarà sull'applicazione dei pattern Aggregate ed EventSourcing.
2. Verrà analizzato nel dettaglio il funzionamento degli eventi.
3. Seguirà quindi una sezione in cui si presenta un caso d'uso dell'applicazione.

UNIVERSITÀ DEGLI STUDI FIRENZE

ArtworkMicroservice

Un artwork è caratterizzato da: ID dell'utente che lo riporta, latitudine, longitudine, nome, stile, tipo.
Si descrive dapprima il funzionamento di questo servizio ad alto livello.

- **Creazione e modifica:** Un artwork è un aggregato che viene persistito come una lista di eventi in un event store.

  1. La creazione corrisponde ad una richiesta *ArtworkReportNewRequest*, che genererà un opportuno evento.
  2. Un update corrisponde ad una richiesta specifica es. ArtworkUpdateStyleRequest, che aggiungerà un evento all'aggregato.
  3. Una cancellazione corrisponde ad una richiesta gestita come un aggiornamento.

ArtworkMicroservice

- **Prelievo:** Un aggregato artwork viene ricostruito attraverso l'applicazione di una lista di eventi da un event store.
  1. Viene creato un **nuovo** aggregato con l'opportuno id.
  2. Vengono caricati ed applicati tutti gli eventi dall'event store riferiti all'aggregato con quell'id.
  3. Il nuovo aggregato viene restituito.

UNIVERSITÀ DEGLI STUDI FIRENZE

ArtworkMicroservice

Prima di analizzare il funzionamento si introducono i cambiamenti principali rispetto agli altri servizi, ovvero quei packages specifici per questo servizio.

- **Aggregates**: Package che implementa un aggregato, il quale è composto da un *aggregate root astratta* e un *aggregato concreto* che implementa la root astratta (in questo caso l'artwork), la root ha i seguenti compiti:
  1. Gestisce l'ID univoco (si usa UUID).
  2. Gestisce gli eventi:

```
private UUID id;
private List<Event> events = new ArrayList<Event>();
private int version;
```

  3. Gestisce la generazione e "consumazione" attraverso i metodi:
     1. generateEvent(Event e) : genera un evento.
     2. RebuildAggregateStatus(List $<$ *Event* $>$ history): applica la lista degli eventi dalla root all'aggregato concreto.
     3. ApplyEvent(Event event, boolean isNew): applica un evento se è nuovo.

ArtworkMicroservice

L'**applicazione di un evento** va dalla root astratta all'aggregato concreto, questo avviene nel seguente modo:

1. La root astratta definisce una serie di eventi astratti che saranno implementati dall'aggregato:

```
public abstract void apply(ArtworkNameChangedEvent event);
```

2. Si usa quindi un metodo (per evitare la reflection) che se richiamato a runtime dalla root (Riferendosi all'aggregato concreto) permette di applicare un evento:

```
private void invokeApplyMethodNoReflection(Event e) {
    if (e instanceof ArtworkNameChangedEvent) {
            this.apply((ArtworkNameChangedEvent) e);
    }
    if (e instanceof ArtworkCreatedEvent) {
            this.apply((ArtworkCreatedEvent) e);
    }
    //...

}
```

ArtworkMicroservice

L'aggregato concreto:

1. Estende la root astratta:

```
public class ArtworkAggregate extends AggregateRoot
```

2. Implementa tutti i campi dell'artwork:

```
private String name;
private ArtStyle style;
...
```

3. Gestisce i vari eventi, ad esempio l'evento di cambio autore:

```
public void changeAuthor(Author newAuthor) {
    this.artworkCreator = newAuthor;
    generateEvent(new ArtworkAuthorChangedEvent(getId(), newAuthor));
}
```

ArtworkMicroservice

Si descrive adesso la gestione degli eventi (per poi passare all'event store).

- Un **Evento** è una classe astratta che ha la seguente struttura:

```
public abstract class Event {
    public UUID aggregateId;
    public int eventNumber;
}
```

- L'**evento concreto** veicola un operazione, esempio nel caso di cambio di autore, si avrà un campo dentro l'evento

```
public Author newAuthor;
```

  Che verrà poi gestito dall'aggregato concreto con l'opportuno apply, esempio:

```
public void apply(ArtworkAuthorChangedEvent event) {
    this.artworkCreator = event.newAuthor;
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

ArtworkMicroservice

Si descrive adesso il funzionamento dell'**event-store**, quest'ultimo è un package che contiene la classe "SimpleEventStore" che:

1. Implementa la funzionalità di event store.
2. E' una classe singleton.
3. Gestisce una concorrenza "di base".

ArtworkMicroservice

L'event store nel suo nucleo è una **mappa concorrente**:

```
private Map<UUID, List<Event>> events = new ConcurrentHashMap<>();
```

A cui si applica un interfaccia con applicazioni di base, simili ad un DBMS:

1. **save**: persiste un aggregato (lista di eventi riferita ad un id).

   ```
   events.put(uuid, newEvents);
   ```

2. **load**: restituisce una lista di eventi.

   ```
   events.get(aggregateId);
   ```

ArtworkMicroservice

L'ultima caratteristica di ArtworkMicroservice che lo distingue dagli altri servizi è la capacità di **richiedere entità** (come DTOs).

- Questo si realizza attraverso degli opportuni Clients (Jax-Rs) nell'opportuno package "Clients"
- Si prevedono due clients, uno per il microservizio degli utenti, ed uno per gli autori.

Caso d'uso

Si riporta un potenziale **caso reale** che descrive l'interazione tra l'utente ed il sistema.
Si descriveranno i seguenti eventi:

1. Un artwork viene segnalato da un certo utente.
2. Lo stesso artwork viene modificato, ad esempio ne viene aggiornato lo stile.
3. L'artwork viene cancellato.

UNIVERSITÀ DEGLI STUDI FIRENZE

Caso d'uso - Creazione

La **creazione** di un utente consiste in una richiesta (con contratto dati json) di tipo

POST http://localhost:8080/AuthorMicroservice/rest/AuthorMicroservice/

Con il seguente corpo:

```
{
        "name" : "Graffito di via X",
        "style" : "COMIC",
        "type" : "MURALES",
        "latitude" : 210,
        "longitude" : 34,
        "reportingUserID" : 1,
        "artworkCreatorID" : 9
}
```

Il servizio risponde creando l'artwork e resituendo il relativo DTO (con in relativo UUID, es: 27d1fa20-9240-49ec-9dee-87de53e27a3e).

Caso d'uso - Update e cancellazione

Si suppone che un utente desideri **cambiare** il tipo di artwork, modificandolo da "MURALES" a "STICKER", si eseguirà una richiesta:

PATCH http://localhost:8080/.../ArtworkMicroservice/updatetype/{id}

con ID: 1fa20-9240-49ec-9dee-87de53e27a3e, e con relativo corpo:

```
{
    "type" : "STICKER"
}
```

Il sistema aggiungerà l'evento opportuno, restituisce poi il relativo DTO a conferma della avvenuta modifica.
Allo stesso modo la cancellazione sarà un opportuna richiesta

DELETE http://localhost:8080/.../ArtworkMicroservice/{ID}

UNIVERSITÀ DEGLI STUDI FIRENZE

Caso d'uso - GET

Si definiscono **3 tipologie** di richieste get:

1. GET che restituisce **tutti gli artworks** come *List* $<$ *ArtworkDTO* $>$:

   GET http://localhost:8080/.../ArtworkMicroservice/

2. GET che restituisce **solo un artwork**:

   GET http://localhost:8080/.../ArtworkMicroservice/{id}

3. GET che restituisce una **"view storica"** di un Artwork, ovvero una lista ordinata dei suoi eventi, questo corrisponde alla necessità di storicizzazione.

Si riportano quindi due esempi, relativi alle get (2) e (3)

Caso d'uso - GET

Esempio di restituzione di un Artwork (ArtworkDTO):


```
{
        "author_id": 10,
        "id": "27d1fa20-9240-49ec-9dee-87de53e27a3e",
        "latitude": 210,
        "longitude": 34,
        "name": "Graffito Sovrascritto",
        "style": "COMIC",
        "type": "STICKER",
        "user_id": 1
}
```

Caso d'uso - GET

Esempio di restituzione di un Artwork come **lista di eventi**:

GET http://localhost:8080/.../ArtworkMicroservice/gethistory/{ID}


```
{
        "eventName": "class events.ArtworkCreatedEvent",
        "version": 1
},
{
        "eventName": "class events.ArtworkStyleChangedEvent",
        "version": 2
},
{
        "eventName": "class events.ArtworkTypeChangedEvent",
        "version": 3
},
{
        "eventName": "class events.ArtworkAuthorChangedEvent",
        "version": 4
},
```

Testing

Tutti e 3 i microservizi hanno una specifica **suite di test** realizzata con Mockito e Junit, ciò che si fa è testare i *controllers* nelle varie funzionalità.

La descrizione della suite di test è omessa nella presentazione (è presente nella documentazione del progetto).

UNIVERSITÀ DEGLI STUDI FIRENZE

Conclusioni e osservazioni

Una prima osservazione riguarda la compatibilità dell'architettura con DBMS diversi.

- In **UserMicroservice** e **AuthorMicroservice** non si incontrano particolari problemi, la gestione della persistenza è delegata a JPA pertanto sarà sufficente modificare il file "persistence.xml" per interagire con un DBMS diverso, es. PostgreSQL.

  Hibernate consente anche l'interazione con database noSQL o basi di dati in-memory.

- Diversa è la questione in **ArtworkMicroservice**, in quanto attualmente la base di dati è un event-store in-memory.

UNIVERSITÀ DEGLI STUDI FIRENZE

Conclusioni e osservazioni

Supponendo che **non** si possa modificare la natura event-sourcing based, è necessario definire un modo in cui si possa implementare o *adattare* l'event-store ad un database relazionale. Si possono prevedere più strategie:

1. Persistere la mappa dell'event-store.
2. Costruire un event-store in approccio OOP e delegare la persistenza a JPA.
3. Utilizzare un framework che esegua automaticamente la persistenza.

UNIVERSITÀ DEGLI STUDI FIRENZE

Conclusioni e osservazioni

1. Persistere la mappa dell'event-store:
   Si persiste la mappa che è alla base dell'EventStore stesso (non consigliato).
2. Costruire un event-store in ottica OOP e delegare la persistenza a JPA:
   Il nucleo di questa implementazione consiste nel creare una serie di oggetti con relative associazioni, esempio una classe "Events" che contiene:
   1. Un riferimento all'id dell'aggregato target dell'evento.
   2. Il riferimento ad una lista di Eventi mappata come collection.

   Una classe "entities" per la gestione delle entità (es. versione, id, tipo) e una classe "snapshot" per la gestione delle snapshot.
3. Utilizzare un framework che esegua automaticamente la persistenza (Axon, Eventuate Local).

Conclusioni e osservazioni

Un ulteriore opzione può essere quella di costruire un Event Store su RDBMS in modo ”manuale”.
Richardson nel suo frame Eventuate Local segue questa opzione, l’event-store viene persistito in un database (MySQL 8.0) con la seguente struttura [18]:

Event database

EVENTS

event_id	event_type	entity_type	entity_id	event_data
102	Order Created	Order	101	{...}
103	Order Approved	Order	101	{...}
...	...	...	...	...

ENTITIES

entity_type	entity_id	entity_version	...
...	...	...	...

SNAPSHOTS

entity_type	entity_id	entity_version	...
...	...	...	...

UNIVERSITÀ DEGLI STUDI FIRENZE

Osservazione sull'architettura

Un osservazione importante riguarda l'architettura, un sistema orientato a **microservizi** si è rivelato essere estremamente flessibile e capace di gestire casi d'uso molto diversi.
Rimangono tuttavia delle problematiche da tenere in considerazione:

- **Latenza**: Un architettura che massimizza la distribuzione sulla rete può portare problemi di latenza.
- **Comunicazione sincrona:** La comunicazione generalmente è sincrona.
- **Consistenza dei dati:** Questo è forse il problema maggiore, se si aggiornano uno o più campi di entità nel servizio A, questi aggiornamenti dovrebbero essere propagati a tutti i servizi che interagiscono (anche) con entità del servizio A.
- **Decomposizione in microservizi:** In alcuni domini può essere difficile dividere in microservizi ad esempio sistemi che contengono classi god [18].

UNIVERSITÀ DEGLI STUDI FIRENZE

Osservazioni sui pattern

CQRS si è dimostrato *fondamentale* in tutti i microservizi per:

1. Dare un *ordine logico* ai vari componenti funzionali, dividendo le operazioni CUD dalle read (R).
2. Rendere *scalabile* il sistema, riducendo il carico di lavoro su un unico endpoint e fornendo le basi per ”viste” sui dati che possono essere rese personalizzate.
3. *Separare le responsabilità*, attraverso CQRS è risultato più facile identificare e risolvere bugs.

   Le richieste CUD, implementati come oggetti rendono facilmente gestibili eccezioni (es. risorse malformate) e gli eventi.

CQRS a mio avviso è da utilizzare sempre dove possibile, tuttavia in domini più complessi può essere richiesta la presenza di più di un database (Scrittura e lettura), con relativi problemi di consistenza e *agreement*.

UNIVERSITÀ DEGLI STUDI FIRENZE

Osservazioni sui pattern

Aggregate diviene fondamentale a mio avviso per domini complessi indipendentemente dai pattern che si andranno ad applicare:

1. Permette di gestire più facilmente cluster di oggetti, per via della presenza dell' *aggregate root*.
2. Ha un risvolto positivo sulla *security* del programma.

   Questo perchè operazioni *critiche* che interessano tutto il cluster possono essere svolte proprio sulla root, minimizzando la superficie di rischio e quindi di fallimento.

CQRS e Aggregate sono quindi ottimi per sicurezza, scalabilità e gestione a livello logico dell'architettura e andrebbero utilizzati sempre dove possibile.

UNIVERSITÀ DEGLI STUDI FIRENZE

Osservazioni sui pattern

L'Implementazione del pattern **Event Sourcing** invece spesso può non essere conveniente, perchè:

1. La procedura di "ricostruzione" di un aggregato dati i suoi eventi ha un costo *lineare* rispetto al numero di eventi stessi.
2. Il numero di eventi tende a crescere rapidamente, può nascere il bisogno di gestire *eventi combinati* (che sono a tutti gli effetti dei nuovi eventi).
3. Gli eventi possono essere soggetti a cambiamenti nel futuro (in nome, struttura e parametri).

   Questo può aggiungere un ulteriore nota di complessità nell'aggiornamento del sistema, soprattutto nella retro-compatibilità con i vecchi eventi.

Osservazioni sui pattern

4. EventSourcing può creare problemi di *affidabilità*.

   Supponendo che un aggregato abbia *n* eventi, ed uno di questi sia corrotto si può avere una situazione in cui l'oggetto viene comunque ricostruito ma si perda il controllo sulla sua reale correttezza.

5. EventSourcing infine è un pattern che richiede un *cambiamento di visione* notevole, ha una learning curve a crescita più lenta rispetto ad altri pattern, e può essere difficilmente applicabile in sistemi già esistenti.

6. Dal punto di vista prettamente ”enterprise” non sono presenti implementazioni di sistemi che usano EventSourcing utilizzate su larga scala e applicate su sistemi rilevanti.

UNIVERSITÀ DEGLI STUDI FIRENZE

Conclusioni

Grazie dell'attenzione.

UNIVERSITÀ DEGLI STUDI FIRENZE

Riferimenti I

[1] Schaffer André. *Event Sourcing and CQRS Examples*.

[2] Asc-Lab. *asc-lab/java-cqrs-intro: Examples of implementation CQRS with Event Sourcing - evolutionary approach*. URL: https://github.com/asc-lab/java-cqrs-intro.

[3] Baeldung. *Event Sourcing In Java*.

[4] Baeldung. *spring-event-sourcing-and-cqrs*.

[5] *bliki: CQRS*. URL: https://martinfowler.com/bliki/CQRS.html (visited on 12/20/2021).

[6] Pontet Cedric. *Agile Partner's CQRS*.

[7] *Chapter 26. Exception Handling*. URL: https://docs.jboss.org/resteasy/docs/2.3.3.Final/userguide/html/ExceptionHandling.html (visited on 01/11/2022).

[8] *CQRS and Event Sourcing Intro For Developers*. July 2020. URL: https://altkomsoftware.pl/en/blog/cqrs-event-sourcing/.

[9] Saurabh Dashora. *Three important patterns for building microservices - dzone microservices*. June 2019. URL: https://dzone.com/articles/3-most-important-patterns-for-building-microservic.

UNIVERSITÀ DEGLI STUDI FIRENZE

Riferimenti II

[10] Oracle Documentation. *Extracting Request Parameters (The Java EE 6 Tutorial)*. docs.oracle.com. URL: https://docs.oracle.com/cd/E19798-01/821-1841/gipyw/index.html.

[11] *Event Sourcing*. URL: https://martinfowler.com/eaaDev/EventSourcing.html (visited on 12/20/2021).

[12] *Eventuate*. URL: https://eventuate.io/.

[13] Martin Fowler. *Bliki: Ddd$_a$ggregate*. Apr. 2013. URL: https://martinfowler.com/bliki/DDD_Aggregate.html.

[14] Mincong Huang. *Exception Handling in JAX-RS*. en. Dec. 2018. URL: https://mincong.io/2018/12/03/exception-handling-in-jax-rs/ (visited on 01/11/2022).

[15] Kiran Kumar. *Microservices with CQRS and event sourcing - dzone microservices*. May 2020. URL: https://dzone.com/articles/microservices-with-cqrs-and-event-sourcing.

[16] Vinicius Feitosa Pacheco. *Microservice Patterns and Best Practices: Explore Patterns like CQRS and Event Sourcing to Create Scalable, Maintainable, and Testable Microservices*. Packt Publishing, 2018. ISBN: 1788474031.

UNIVERSITÀ DEGLI STUDI FIRENZE

Riferimenti III

[17] *RESTEasy Exception Handling with ExceptionMapper— JBoss.org Content Archive (Read Only)*. URL: https://developer.jboss.org/docs/DOC-48310 (visited on 01/11/2022).

[18] *Richardson Maturity Model*. URL: https://martinfowler.com/articles/richardsonMaturityModel.html (visited on 01/11/2022).

[19] *What are microservices?* URL: http://microservices.io/index.html (visited on 12/20/2021).

[20] *Why would I need a specialized Event Store? - AxonIQ*. URL: https://axoniq.io/blog-overview/eventstore (visited on 01/20/2022).

[21] John Au-Yeung and Ryan Donovan. *Best practices for REST API design*. en-US. Mar. 2020. URL: https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/ (visited on 01/11/2022).

[22] Greg Young. "CQRS, Task Based UIs, Event Sourcing agh!" In: *codebetter* (2010). URL: https://gist.github.com/meigwilym/025f08208b5640ad26bc410c8a83b10f.