Documentazione Elaborato SWAM

Implementazione di un architettura software back-end a *microservizi* orientata agli eventi con *CQRS* ed *EventSourcing* per l'applicazione StreetApp.

Alessandro Mini - 7060381

Elaborato del corso di Software Architecture and Methodologies
(9CFU)
2021-2022

CdL Magistrale di Ingegneria Informatica
Università degli Studi di Firenze

Indice

1 Introduzione e Analisi dei requirements **3**
- 1.1 Requisiti Funzionali . . . 3
- 1.2 Requirements non funzionali . . . 4
- 1.3 Obiettivi architetturali . . . 4

2 Domain Model **5**

3 Dettaglio sulle tecnologie **7**
- 3.0.1 Tecnologie per lo sviluppo . . . 7

4 Implementazione (Abstract) **8**
- 4.0.1 Dettaglio sui pattern implementati . . . 9
  - 4.0.1.1 CQRS . . . 9
  - 4.0.1.2 EventSourcing . . . 10
  - 4.0.1.3 Aggregate . . . 10
- 4.0.2 Gestione delle eccezioni . . . 11
- 4.1 Dettaglio sulla comunicazione . . . 12
- 4.2 Identificazione degli eventi . . . 12

5 Implementazione (Java) **13**
- 5.1 UserMicroservice . . . 14
  - 5.1.1 Metodi Implementati . . . 14
  - 5.1.2 Architettura . . . 15
  - 5.1.3 Flusso dei dati . . . 19
- 5.2 AuthorMicroservice . . . 23
  - 5.2.1 Metodi Implementati . . . 24
  - 5.2.2 Architettura . . . 24
  - 5.2.3 Flusso dei dati . . . 26
- 5.3 ArtworkMicroservice . . . 29
  - 5.3.1 Metodi Implementati . . . 30
  - 5.3.2 Architettura . . . 31

6 Implementazione (Basi di dati) **40**
- 6.0.1 Basi di dati . . . 40
- 6.0.2 Base di dati UserMicroservice . . . 40
- 6.0.3 Base di dati AuthorMicroservice . . . 40

7 Analisi di funzionamento **41**

8 Testing **45**
- 8.1 Test in UserMicroservice . . . 45
  - 8.1.1 Test Command Endpoint . . . 45
  - 8.1.2 Testing View Controller . . . 46
- 8.2 Test in AuthorMicroservice . . . 46
  - 8.2.1 Test Command Endpoint . . . 46
  - 8.2.2 Testing View Controller . . . 47

8.3 Test in ArtworkMicroservice 47
8.3.1 Test Command Controller 47
8.3.2 Testing View Endpoint 48
8.3.3 Testing SimpleEventStore 49

9 Osservazioni e Conclusioni **50**

1 Introduzione e Analisi dei requirements

In questo progetto si implementa il back-end dell'app StreetApp, un app social per segnalare, riportare e geo-localizzare **graffiti**.

Il progetto riguarda la gestione degli *artworks*, un *artwork* (prodotto artistico) è un artefatto che deve essere gestito dal sistema, è identificato 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 ...)

I prodotti artistici sono *segnalati* dagli utenti, un utente può gestire le proprie segnalazioni e informazioni.

C'è necessità di *storicizzazione* degli artwork, in quanto un graffito potrebbe essere coperto da uno successivo.

1.1 Requisiti Funzionali

Si è analizzato il problema partendo dai suoi requisiti, quelli **funzionali** principali identificati sono:

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. Segnalazione di un artwork da parte di un utente: Un Artwork è segnalato da un utente.
4. Modifica di un artwork di tipo *"standard"*: Viene modificato il nome di un Artwork, o alcune sue caratteristiche.
5. Modifica di un artwork *"drastica"*: 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: Poter eseguire report "storicizzati" in cui sia visibile la lista degli eventi precedenti di un certo artwork.
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*.

   Si identifica quindi un *subset* di campi "obbligatori" per la creazione di un Artwork che sono: latitudine, longitudine, nome e l' ID dell'utente che lo riporta.

   Tutti gli altri potranno essere aggiunti in seguito.

1.2 Requirements non funzionali

I requisiti **non-funzionali** principali identificati sono:

1. Gli artwork sono identificati univocamente da un ID, latitudine e longitudine i quali **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).

1.3 Obiettivi architetturali

L'obiettivo di questo progetto è quello di andare ad implementare i tre servizi sopra descritti per costruire un sistema back-end ibrido che combini l'approccio classico SOA con quello *event driven*.

Si andranno ad implementare pattern specifici, rimanendo nella "compliance" ai requisiti del paradigma architetturale **REST**.

L'implementazione consiste in appunto tre microservizi, ognuno dei quali esporrà degli specifici metodi HTTP, nel dettaglio i microservizi sono i seguenti:

1. *UserMicroservice*: Servizio per la gestione degli utenti, con il proprio DBMS Mysql 8.0 e gestione delle richieste disaccoppiata mediante il pattern CQRS.
2. *AuthorMicroservice*: Servizio per la gestione degli autori, anch'esso con il proprio DBMS Mysql 8.0 e gestione delle richieste disaccoppiata mediante il pattern CQRS.
3. *ArtworkMicroservice*: Servizio per la gestione degli artworks, dal punto di vista della memorizzazione si utilizza il pattern EventSourcing pertanto sarà dotato di un event-store.

   In questo servizio oltre al pattern CQRS si implementa anche *EventSourcing* ed *Aggregate*, la cui spiegazione verrà data in seguito.

Questi servizi, così come gli eventi, sono stati individuati attraverso un processo di analisi **DDD** (section 2.)

Un altro obiettivo di questo progetto è quello di andare a costruire un adeguata suite di *test*, in cui si andranno a verificare le funzionalità principali, attraverso i frameworks *Junit e Mockito*.

Seguirà infine un caso d'uso in cui si mostrano le funzionalità del sistema in un caso reale (section 7)

2 Domain Model

Un primo passo nell'approcciarsi al problema è di andare ad analizzare il domain model, grazie al quale si è rivelato immediato suddividere la potenziale architettura in microservizi.

Ho deciso di seguire l'approccio DDD presente in [16], in cui si propone di partire da una definizione di *domain model* "globale" per poi identificare i *bounded contexts* e conseguentemente i potenziali *microservizi*, seguendo poi una fase finale di raffinamento dei singoli domain models.

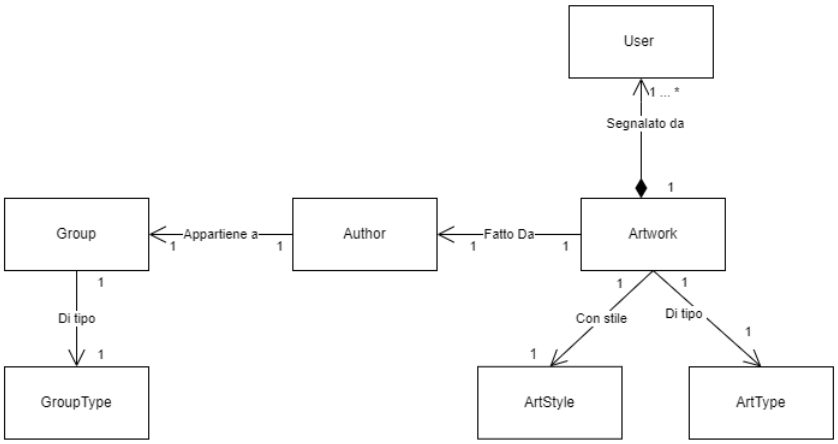


Figura 1: Domain Model identificato (completo).

Quindi ho identificato i bounded contexts:

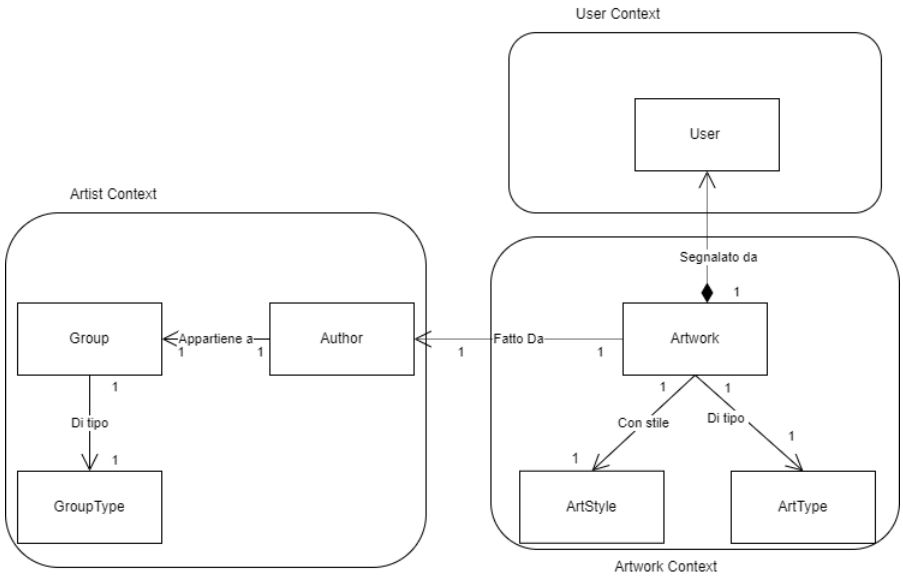


Figura 2: Bounded contexts identificati.

Dai 3 bounded context si sono identificati i 3 servizi *UserMicroservice, AuthorMicroservice, ArtworkMicroservice.*, nel dettaglio, dal punto di vista di modellazione del dominio:

- **UserMicroservice** Implementa soltanto la domain logic dell'User (in linea di principio potrebbe implementare anche la logica di accesso alle varie risorse, quindi l' AAA (autenticazione, autorizzazione, accounting).
- **AuthorMicroservice** Implementa il contesto dell'autore, gestisce quindi quest'utlimo, il gruppo (CREW o Movimento) e la relativa appartenenza.
- **ArtworkMicroservice** Implementa l'artwork, interagisce con gli altri servizi secondo i principi dello stile architetturale REST per reperire lo stato di utenti/autori.

  Gestisce gli artwork e le sue caratteristiche (lat,long,nome,stile,tipo) e le relazioni con gli altri elementi (Artwork riportato da user, e creato da autore) con relativi updates storicizzati e memorizzati con event sourcing.

3 Dettaglio sulle tecnologie

L'implementazione del progetto consiste in tre progetti Eclipse , uno per microservizio.

3.0.1 Tecnologie per lo sviluppo

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 .
- GIT.
- Insomnia.

Per il testing ho utilizzato:

- Junit versione 4.12.
- Mockito versione 4.2.0.

L'analisi dell'implementazione partirà dai servizi più semplici, analizzando prima l'architettura generale e poi alcuni dettagli implementativi, verranno omessi i file di configurazione (come il file persistence.xml).

4 Implementazione (Abstract)

In questa sezione si analizza l' **architettura complessiva** e la suddivisione delle varie entità nei vari servizi, si passerà poi al dettaglio implementativo di quest'ultimi.

Ogni servizio implementato avrà una struttura simile, gli elementi principali sono suddivisi in vari *packages*, che saranno presenti nei vari progetti:

- *controllers*: Package che implementa i vari controller specifici per un certo servizio.
- *DAOs e DTOs:* Sono due packages distinti che implementano rispettivamente i DAOs e i DTOs, tra i DTOs sono presenti quelli relativi alle richieste (requests) che vengono inviate al sistema.
- *endpoints*: Package che implementa gli endpoints (classi anntoate jax-rs), secondo il pattern CQRS ogni servizio avrà due endpoints, uno per la gestione dei commands ed uno per la gestione delle views, rispettivamente operazioni CUD ed operazioni R.
- *entities*: Conterrà le classi relative al domain model dello specifico servizio in analisi.
- *exceptions*: Contiene delle eccezioni specifiche per i servizi, le eccezioni sono gestite attraverso l'utilizzo di *ExceptionMapper* di jax-rs.
- *mappers*: Contiene i mappers per la conversione tra DTOs ed entities.
- *settings*: Package specifico per ogni servizio che implementa le configurazioni.

Il servizio ArtworkMicroservice conterrà altri elementi, infatti avrà i packages:

- *events*: Package che gestisce gli eventi (evento astratto ed eventi concreti per le varie operazioni che si possono eseguire sugli artworks).
- *eventstore*: Contiene le classi releative al "SimpleEventStore", un event-store in-memory concepito per realizzare il pattern EventSourcing.
- *aggregates*: Package che implementa il pattern Aggregate, questo pattern è implementato soltanto in ArtworkMicroservice perchè diviene funzionale all'implementazione del pattern eventSourcing.
- *clients*: Package che contiene i client Jax-rs per richiedere le entità agli altri servizi.

Segue quindi una descrizione delle *entità principali*, ovvero quelle relative al domain model (e non relative al funzionamento dell'architettura come ad esempio le classi relative agli eventi).

Le entità principali che vengono implementate sono le seguenti:

- *User*: Entità che rappresenta l'utente dell'applicazione, un User è identificato dal proprio id e dal proprio username, che è soggetto a vincolo di integrità.
- *Author*: Entità che rappresenta l'autore di un artwork, un autore sarà identificato dal proprio id e dal suo nome, che è soggetto a vincolo di integrità, dal nome del gruppo a cui appartiene ed infine al tipo del suddetto gruppo.

- *GroupType*: Entità (concretamente, una enum) che descrive le tipologie di gruppo a cui un autore può appartenere, ad esempio una CREW oppure un MOVEMENT.
- *ArtworkAggregate*: Rappresenta un artwork, questo viene implementato secondo il pattern aggregate.
- *ArtStyle* : Entità (enum) che rappresenta lo *stile* di un artwork.
- *ArtType* : Entità (enum) che rappresenta il *tipo* di un artwork.

La suddivisione di queste entità nei vari servizi rispecchia quella dei relativi *bounded contexts* come in Figure 2.

4.0.1 Dettaglio sui pattern implementati

I pattern implementati sono *CQRS, EventSourcing, Aggregate, Singleton.*

In questa sottosezione si riporta il funzionamento e l'implementazione a livello generale di questi patterns, per brevità si omette il funzionamento del pattern "singleton".

4.0.1.1 CQRS

Nella versione di **CQRS** implementata ci si limita al *solo disaccoppiamento* delle possibili operazioni tra operazioni CUD (commands) e operazioni read (views).

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

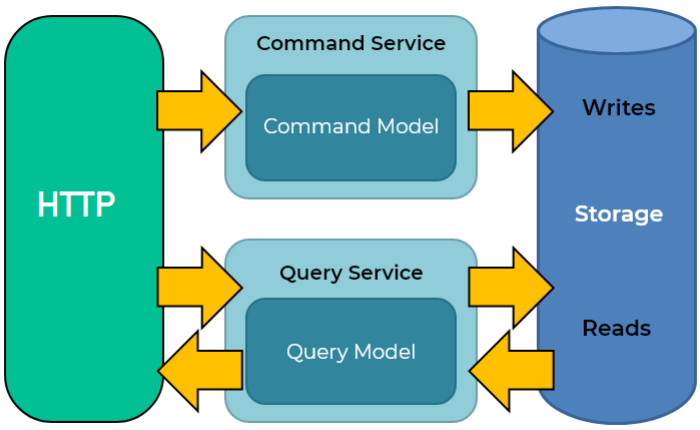


Figura 3: Applicazione di CQRS nel sistema.

Per le scritture si avrà il flusso:

Utente → *CommandEndpoint(request)* → *CommandController(request)* → *Base di dati*

Per le letture si avrà il flusso:

Utente → *ViewEndpoint(request)* → *ViewController(request)* → *Base di dati*

La base di dati relativa a letture e scritture è la **stessa**.

CQRS in alcune implementazioni prevede la presenza di due basi di dati distinte (una per lettura ed una per scrittura), in questa implementazione ne è presente soltanto uno (che coincide), 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 (e.g. il raggiungimento dell'*agreement* sulle transazioni).

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].

4.0.1.2 EventSourcing

Il pattern EventSourcing prevede di:

1. Gestire un entità attraverso i suoi eventi, ogni modifica all' entità 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.

L'applicazione di questo pattern è estremamente efficiente per la gestione della *storicizzazione* delle entità, in quanto queste saranno fondamentalmente una "lista di eventi".

L'implementazione di questo pattern richiede la presenza di un *event-store*, una base di dati orientata agli eventi che può essere genericamente vista come una *mappa* del tipo:

$$M < A, [e_1, e_2, \ldots, e_n] >$$

Che ad un certo aggregato A (aggregate root A) associa una lista di eventi $e_1 ... e_n$.

4.0.1.3 Aggregate

Il pattern **Aggregate** [13] viene implementato in *ArtworkMicroservice* per la gestione di un Artwork, che è un aggregato.

Un Aggregate rappresenta un *cluster di oggetti* che possono essere trattati come una singola unità, è identificato da una **root**, un oggetto (nel progetto sarà una classe astratta) che garantisce l'integrità ed espone delle operazioni di base.

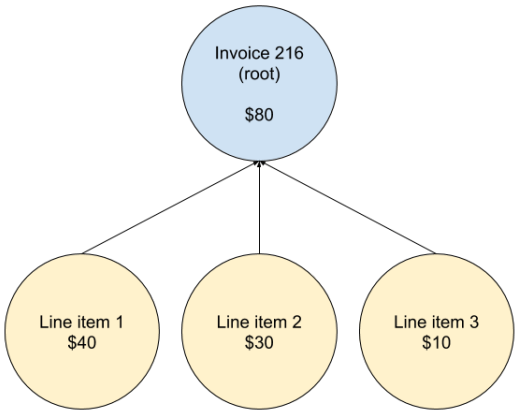


Figura 4: Esempio (grafico) dell'applicazione del pattern *aggregate*, le operazioni saranno invocate sulla root.

Nell'ottica EventSourcing l'aggregate root è *responsabile della gestione degli eventi* che interessano l'aggregate, esporrà quindi metodi per aggiungere un evento, per generarlo e per ricostruire lo stato dell'aggregate a partire dai suoi eventi.

La struttura dell'aggregate pattern è incredibilmente vantaggiosa per la cooperazione con il pattern EventSourcing, questo perchè:

1. Fornisce un punto di centralizzazione per la gestione degli eventi: un evento e viene generato dalla root (e consumato eventualmente da altre entità).

   Un evento inoltre viene applicato dalla root dell'aggregato, la quale gestisce anche le operazioni di *rebuild* che sono critiche ed estremamente vitali per il funzionamento del pattern EventSourcing.

2. Rende più fattibile l'integrazione di diverse tipologie di aggregati, in quanto la gestione degli eventi è simile.

3. Rende più facile il trasferimento e la copia, in quanto l'aggregate root gestisce anche la possibilità di "trasferire tutti gli eventi" ad un altra root di un altro aggregato.

4.0.2 Gestione delle eccezioni

Per la gestione delle eccezioni si è utilizzato il sistema delle exception user-defined e degli *ExceptionMappers* [17], [7], [14], in accordo alla documentazione di RESTEASY e Jax-RS.

Si implementano quindi delle eccezioni di tipo user-defined 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*.

4.1 Dettaglio sulla comunicazione

Ogni microservizio espone dei metodi HTTP secondo lo standard di specifica per le API orientate al paradigma architetturale REST [18], [21].

I servizi saranno in grado di *inter-comunicare*, ad esempio ArtworkMicroservice potrà richiedere informazioni ad UserMicroservice ed AuthorMicroservice.

4.2 Identificazione degli eventi

Dopo aver identificato i microservizi, si procede identificando i potenziali *eventi*.

Gli eventi che si sono identificati riguardano la gestione dell'Artwork Context (la porzione di sistema orientata interamente ad eventi) e sono i seguenti:

1. *ArtworkCreatedEvent* : Evento principale, riguarda la creazione di un artwork e viene innescato dalla segnalazione di un utente di quel determinato artwork.

2. *ArtworkAuthorChangedEvent* : Evento di modifica dell'autore di un artwork.

3. *ArtworkNameChangedEvent* : Evento di modifica del nome di un artwork, modella una situazione del tipo "Un utente X assegna un nome più appropriato all'artwork".

4. *ArtworkRemovedEvent* : L'artwork viene rimosso, questo evento viene inserito tra gli eventi di un artwork per segnalare la sua rimozione, la rimozione non elimina l'artwork nel suo intero dall'event-store, questo perchè potrebbe essere ricreato in futuro, ed è comunque una parte necessaria alla storicizzazione.

5. *ArtworkStyleChangedEvent* : Evento di modifica dello stile di un artwork, modella una situazione del tipo "un nuovo stile artistico viene individuato e si adatta meglio alle caratteristiche dell'artwork".

6. *ArtworkTypeChangedEvent* : Evento di modifica del tipo di un artwork.

5 Implementazione (Java)

I 3 progetti JavaEE 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.

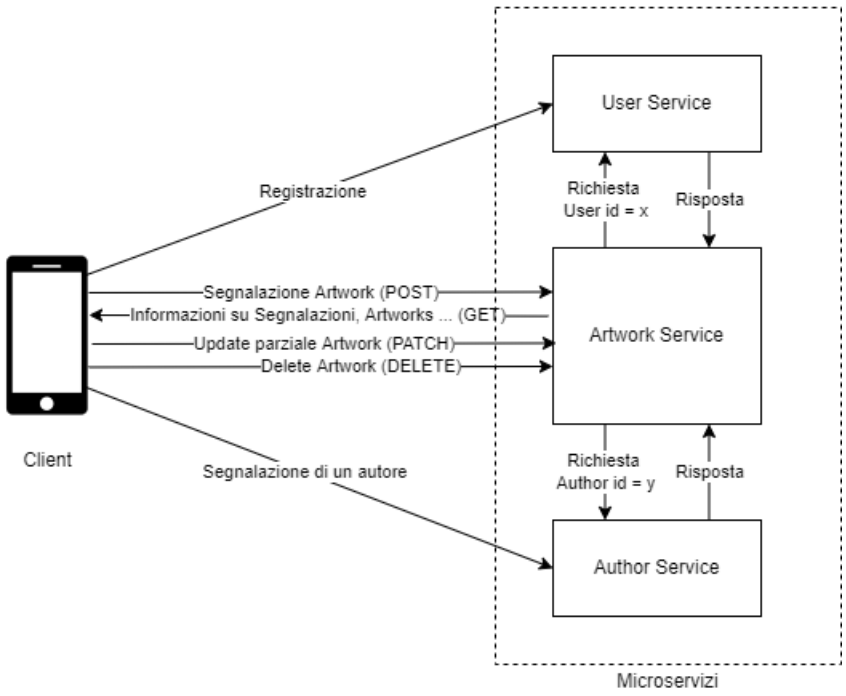


Figura 5: Comunicazione tra utente e servizi allo stato attuale

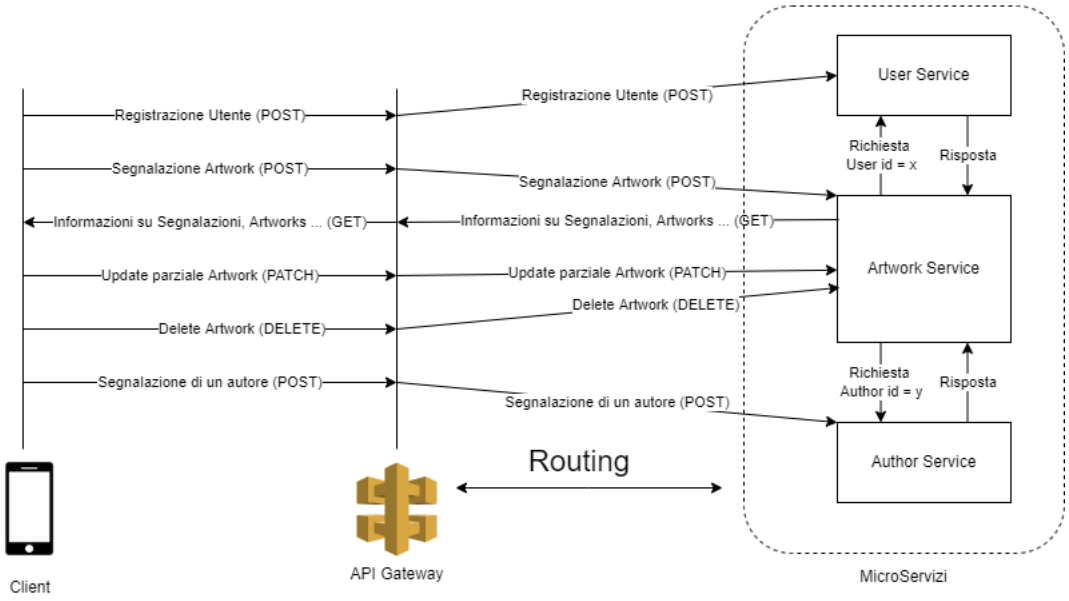


Figura 6: Comunicazione tra utenti e servizi con (potenziale) gateway.

5.1 UserMicroservice

Si inizia l'analisi dal servizio più semplice, ovvero **UserMicroservice**.

5.1.1 Metodi Implementati

Dal punto di vista logico un **user** è caratterizzato dai seguenti campi:

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

Questo servizio espone i seguenti metodi HTTP:

- *POST http://localhost:8080/UserMicroservice/rest/UserMicroservice* : Riceve delle richieste in POST di tipo UserCreationRequest (un utente viene creato), un Utente inoltre è identificato da 2 campi: ID (univoco, gestito da Hibernate) e username che identifica il nome dell'user.
- *PATCH http://localhost:8080/UserMicroservice/rest/UserMicroservice/updateusername/{id}:* Riceve una richiesta di tipo UserUsernameUpdateRequest, è una richiesta di update dell'username.
- *DELETE http://localhost:8080/UserMicroservice/rest/UserMicroservice/{id}*: Richiesta per cancellare un user di tipo DELETE.
- *GET http://localhost:8080/UserMicroservice/rest/UserMicroservice:* Richiesta che restituisce *tutti* gli users.
- *GET http://localhost:8080/UserMicroservice/rest/UserMicroservice/{id}:* Richiesta che restiuisce un user per ID.

Il sistema riceve richieste per le operazione CUD (Create, update, delete) e risponde attraverso DTOs degli users nel caso di chiamate R (GET).

Il flusso delle chiamate di tipo CUD (Commands) e R (Queries) è *disaccoppiato* all'interno del programma.

5.1.2 Architettura

L'architettura del progetto in **eclipse** è la seguente, si descrivono i componenti principali:

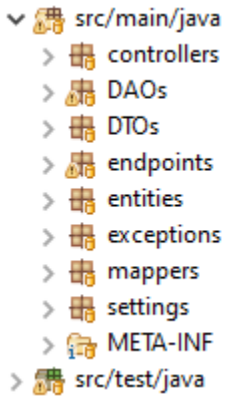


Figura 7: Architettura User Service

- **endpoints**: Sono 2, implementati secondo la logica del pattern CQRS, *UserCommandEndpoint* che riceve comandi CUD da interfaccia HTTP e *UserViewEndpoint* che riceve comandi R.

  L'*UserCommandEndpoint* riceve delle richieste (Oggetti java) in ingresso da protocollo HTTP, le inoltra al controller di competenza, attende i risultati e risponde con degli status opportuni sulla base del fatto che il comando sia stato eseguito con successo o meno.

  L'*UserViewEndpoint* riceve in ingresso delle *richieste* di lettura (queries) da HTTP, le inoltra al controller di competenza, attende i risultati e risponde con degli UserDTO in caso affermativo (l'utente è stato reperito con una certa query) oppure con degli status di errore.

  Esempio di gestione di una richiesta di update in *UserCommandEndpoint*:

```
@Path("/updateusername/{id}")
@PATCH
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response updateUser(@PathParam("id") int id,
↪  UserUsernameUpdateCommandDTO request) throws Exception {
        Response response;
        try {
                User updated = userCommandController.update(id, request);
                response = Response.ok(new
                ↪  UserMapper().UserToDto(updated)).build();
        } catch (ResourceNotFoundException e) {
                throw e;
        } catch (BadRequestException e) {
                throw e;
        }
        return response;
}
```

Un metodo di gestione di una query invece lo si può trovare nel *UserViewEndpoint*:

```
@Path("/{id}")
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response findUserById(@PathParam("id") int id) throws Exception {
        Response response;
        try {
                UserDTO found = userViewController.findById(id);
                response = Response.ok(found).build();
        } catch (ResourceNotFoundException e) {
                throw e;
        }
        return response;

}
```

La versione del pattern **CQRS** implementata per *UserMicroservice* ed *AuthorMicroservice* si limita al solo disaccoppiamento dei modelli di gestione di Commands e Queries, con un unico database in comune, questo è stato analizzato in paragraph 4.0.1.1.

- **DAOs**: contiene la classe UserDAO la quale ha i metodi create, findall, update, delete per interagire con il database attraverso JPA.
- **DTOs**: Contiene il DTO relativo all'user con le seguenti informazioni username, id.

```
public class UserDTO {
        private String username;
        private int id;
        //...
}
```

Questo package contiene anche il DTO della richiesta di creazione di un utente e di update dell'username di un utente, queste richieste vengono implementate come segue:

```
    //Richiesta di creazione di un utente
        public class UserCreationCommandDTO {
            String username;
            public String getUsername() {
                    return username;
            }
            public void setUsername(String username) {
                    this.username = username;
            }
    //...



    }
    //Richiesta di update dell'username di un utente
    public class UserUsernameUpdateCommandDTO {
        String username;

        public String getUsername() {
                return username;
        }

        public void setUsername(String username) {
                this.username = username;
        }
    //...
}
```

I *Commands* (CQRS) sono caratterizzati da richieste, ad esempio di creazione, update o cancellazione di un User (o di un Author, o Artwork negli altri servizi).

Ogni richiesta è una classe Java che veicola tutti i campi necessari per la creazione di un entità (richiesta di creazione) oppure il campo da aggiornare per quella specifica richiesta nel caso di richieste di update.

Questa scelta è stata concepita in modo tale da ottenere una *granularità più fine* nella gestione delle operazioni ammissibili su una certa entità, senza dover gestire richieste di update parziali.

L'elevata granularità che si ottiene con questo principio permette di gestire con maggior accuratezza e semplicità gli eventi e la storicizzazione nell'Artwork Service.

Dato un certo User (in cui sono presenti più campi e non solo l'username) si può essere interessati ad aggiornare solo uno di questi campi, lasciando invariati gli altri.

Nel caso in cui si debba creare un utente, si invierà una richiesta POST con gli opportuni dati all'indirizzo *http://.../create*, il sistema convertirà la richiesta POST in un *UserCreationRequest* che verrà gestita dalla parte Command del pattern CQRS.

Nel caso in cui si debba aggiornare l'username di un utente, si invierà una richiesta PUT di tipo *UserUsernameUpdateRequest* con il valore *"username":"newUsername"*.

Anche in questo caso l'interpretazione della richiesta passerà alla parte Command del pattern CQRS.

Le *queries* allo stato attuale sono delle semplici richieste GET.

- **entities**: 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 **univoco**, in modo tale da evitare User con username duplicato, come nella maggioranza delle applicazioni reali:

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

- **mappers**: 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.

- **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*.

  A titolo di esempio si riporta la gestione di una richiesta di tipo create nell' *UserCommandController*

```
public User create(UserCreationRequest request) throws Exception {
        User u = new User();
        u.setUsername(request.getUsername());
        User created = userDAO.create(u);
        return created;
}
```

- **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.

  Un eccezione di questo tipo ha la struttura:

```
public class ResourceNotFoundException extends Exception implements Serializable {
        private static final long serialVersionUID = 1L;
        public ResourceNotFoundException() {
        }
        public ResourceNotFoundException(String message) {
                super(message);
        }
        public ResourceNotFoundException(String message, Exception e) {
                super(message, e);
        }
}
```

  Con relativo mapper:

```
@Provider
public class BadRequestExceptionMapper implements
↪  ExceptionMapper<BadRequestException> {
        @Override
        public Response toResponse(BadRequestException exception) {
                return Response.status(400).entity(exception.getMessage())
                            .type(MediaType.TEXT_PLAIN).build();
        }
}
```

Nel mapper si può notare la presenza dell'annotazione Provider.

- **settings**: Package simile per tutti i progetti (che verrà omesso d'ora in poi) che contiene le configurazioni RESTEASY.

5.1.3 Flusso dei dati

Si riporta un esempio di funzionamento di richiesta di tipo **Query** (READ) e **Command** (CUD), in accordo al pattern CQRS la gestione di questi due tipi di richieste è disaccoppiata:

- L'utente esegue una richiesta **Query**:

  1. La richiesta viene eseguita attraverso
     GET http://localhost:8080/UserMicroservice/rest/UserMicroservice/{id}
     Questa viene inoltrata da RestEasy al controller (ViewController).
  2. L'endpoint gestisce la richiesta in arrivo reperendo attraverso il controller l'User con id *id*, nel caso di richiesta *GET http://..//* si ha il seguente metodo handler:
     ```
     @GET
     @Produces(MediaType.APPLICATION_JSON)
     public Response findAllUsers() {
             List<UserDTO> found = userViewController.findAllUsers();
             return Response.ok(found).build();
     }
     ```
     Il quale chiama l'userViewController.
  3. L'userViewController risponde attraverso il metodo *findAllusers()* la cui implementazione è :
     ```
     public List<UserDTO> findAllUsers() {
             List<User> users = dao.findAll();
             List<UserDTO> usersDtos = users.stream().map(mapper::UserToDto)
                                        .collect(Collectors.toList());
             return usersDtos;
     }
     ```
     questo metodo prende tutti i dati dal DAO e con un mapper crea una lista di DTOs.
  4. Il metodo *findAllUsers()* interagisce con il dao che restituisce tutti gli users:
     ```
     public List<User> findAll() {
         TypedQuery<User> results = em.createQuery("from User", User.class);
         List<User> queryList = results.getResultList();
         return queryList;

     }
     ```

5. La risposta finale che arriva all'utente con la chiamata GET è strutturata JSON del tipo:

```
[
        {
                "id": 1,
                "username": "test2"
        },
        {
                "id": 20,
                "username": "User11"
        },
]
```

   Questo metodo è semplice e non ha gestione delle eccezioni, nel caso in cui non ci siano users restituisce una lista vuota, la chiamata che prende un user dato l'id gestisce invece le eccezioni, brevemente nel *ViewEndpoint* si ha un metodo:

```
@Path("/{id}")
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response findUserById(@PathParam("id") int id) throws Exception {
        Response response;
        try {
                UserDTO found = userViewController.findById(id);
                response = Response.ok(found).build();
        } catch (ResourceNotFoundException e) {
                throw e;
        }
        return response;

}
```

- Nel caso in cui la richiesta si di tipo **Command** (supponiamo una richiesta create) il flusso è il seguente:

  1. L'utente crea una richiesta *POST http://..//* in cui invia (con contratto json) i dati necessari a popolare una *UserCreationRequest* ovvero soltanto l'*username* (perchè l'ID è gestito da JPA).

     ```
     {
          "username": "newUser"
     }
     ```

  2. RESTEasy invoca l'*UserCommandEndpoint*, nello specifico il metodo di creazione:

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response createUserFromJsonRequest(UserCreationCommandDTO request)
↪  throws Exception {
        Response response;
        try {
                User created = userCommandController.create(request);
                response = Response.ok(new
                ↪  UserMapper().UserToDto(created)).build();
        } catch (BadRequestException e) {
                throw e;
        }
        return response;

}
```

Il quale come si può vedere prende in ingresso una richiesta, invoca quindi il metodo *create* dell'*UserCommandController*.

Nel caso in cui la richiesta di creazione sia **errata**, risponde con un apposita **eccezione**, in caso di creazione avvenuta con successo restituisce il DTO dell'utente appena creato.

La restituzione del DTO in caso di creazione/update avvenuto con successo si fa perchè permette di:

(a) Predisporre in modo più efficace un infrastruttura di testing.

(b) Essere più aderenti ad un caso reale in cui un *client* (finale) del sistema dopo la creazione di un User potrebbe essere interessato a acquisire subito le informazioni relative alla creazione (ad esempio l'ID) che altrimenti richiederebbero ulteriori richieste GET.

3. L'*UserCommandController* gestisce la richiesta, interagisce con il DAO e crea il nuovo utente, partendo appunto dal command controller si ha il seguente metodo di gestione:

```
public User create(UserCreationCommandDTO request) throws Exception {
        User u = new User();
        u.setUsername(request.getUsername());
        User created = userDAO.create(u);
        return created;
}
```

4. Il DAO esegue un operazione di creazione in cui si esegue anche un check di consistenza.

```
public User create(User user) throws Exception {
        if (user.getUsername().equals("")) {
                throw new BadRequestException("Utente con nome non valido");
        }
        try {
                em.getTransaction().begin();
                em.persist(user);
                em.flush();
                em.getTransaction().commit();
        } catch (PersistenceException e) {
                if (e.getCause() instanceof ConstraintViolationException) {
                throw new BadRequestException("Un utente con l'username inserito
                                                è già presente nel sistema");
                }
        }
        User found = em.find(User.class, user.getId());
        if (found == null) {
        throw new BadRequestException("Impossibile creare l'utente,
                                errore sconosciuto");
        }
        return user;

}
```

5. Alla fine all'utente arriva una risposta OK contente il DTO dell'user creato se la creazione è avvenuta, altrimenti una risposta con l'apposita eccezione.

   A titolo di esempio si esegue la richiesta sopracitata la quale restituisce un 200-OK e nel database si può effettivamente notare l'inserimento:

	id	username
1	1	test2
2	20	User11
3	21	User2
4	22	User3
5	23	newUser

Figura 8: User persistito nel database

Se volessimo fare un **update** di questo user si dovranno fare i seguenti step:

(a) Reperire l'id dell'utente.

(b) Eseguire una richiesta *PATCH http://...//updateusername//* con corpo

```
{
"username" : "nuovoUsername"
}
```

in questo caso:

http://localhost:8080/UserMicroservice/rest/UserMicroservice/23

Che produce il seguente risultato:

	id	username
1	1	test2
2	20	User11
3	21	User2
4	22	User3
5	23	nuovoUsername

Figura 9: Update dell'user.

La gestione delle **eccezioni** come analizzato precedentemente avviene attraverso exception definite ad-hoc e exceptionMapper jax-rs, questo permette di raggiungere un elevata granularità nelle eccezioni, ad esempio:

- Richiesta vuota: si invia la richiesta

  ```
  {
      "username" : ""
  }
  ```

  Il sistema risponde con:

  `400 - "Utente con nome non valido".`
- Si invia una richiesta con username già presente nel sistema, il sistema restituisce:

  `400 - Un utente con l'username inserito è già presente nel sistema`
- Si invia una richiesta di lettura con ID non valido, il sistema restituisce:

  `404 - Utente con id X non trovato`

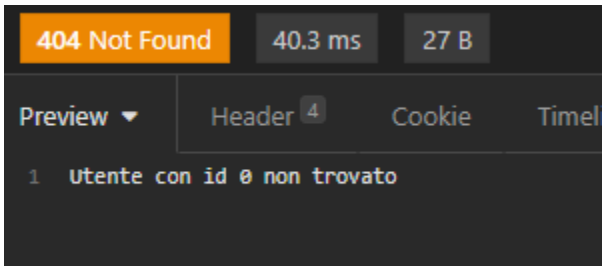


Figura 10: Esempio di errore Risorsa non trovata (404) con client Insomnia.

Metodi simili per la gestione delle eccezioni sono presenti in tutti i servizi.

5.2 AuthorMicroservice

L'implementazione di *AuthorMicroservice* è simile a *UserMicroservice*, il progetto profondamente diverso è *ArtworkMicroservice* in quanto è orientato CQRS+ES, Per AuthorMicroservice si riportano soltanto i cambiamenti rilevanti rispetto ad UserMicroservice.

Un autore ha i seguenti campi:

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

5.2.1 Metodi Implementati

Il servizio espone i seguenti endpoints con relativi metodi:

- POST *http://localhost:8080/AuthorMicroservice/rest/AuthorMicroservice/* metodo di creazione di un author, con contratto dei dati JSON.
- PATCH *http://localhost:8080/AuthorMicroservice/rest/AuthorMicroservice/updatename/*{*id*} metodo di update per cambiare il nome di un autore.
- PATCH *http://localhost:8080/AuthorMicroservice/rest/AuthorMicroservice /updatename/updategroupname/*{*id*} richiesta per cambiare il nome del gruppo di appartenenza di un autore.
- PATCH *http://localhost:8080/AuthorMicroservice/rest/AuthorMicroservice /updatename/updategrouptype/*{*id*} richiesta per cambiare il tipo del gruppo di appartenenza di un autore.
- DELETE *http://localhost:8080/AuthorMicroservice/rest/AuthorMicroservice/*{*id*} richiesta per eliminare un autore dato un ID.
- GET *http://localhost:8080/AuthorMicroservice/rest/AuthorMicroservice/findbyid/*{*id*} richiesta per reperire un AuthorDTO dato un id.
- GET *http://localhost:8080/AuthorMicroservice/rest/AuthorMicroservice/*{*id*} richiesta per reperire *tutti* gli authors del sistema.

5.2.2 Architettura

L'architettura è la seguente:

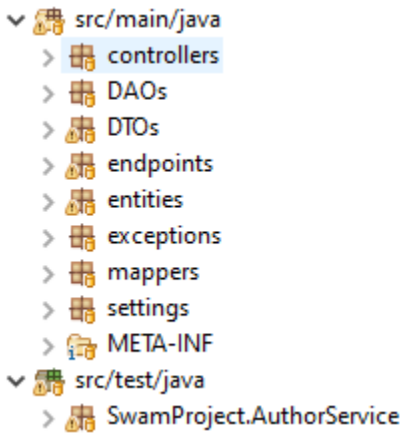


Figura 11: Architettura AuthorMicroservice

La struttura di fondo è simile ad *UserMicroservice*, si riportano i cambiamenti prncipali:

- **Entities**: si hanno 2 entità, l'autore (*Author*) ed il *GroupType* (enum), la struttura di un author prevede le seguenti annotazioni JPA:

```
@Entity(name="Author")
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String authorName;
    private String groupName;

    @Enumerated(EnumType.STRING)
    private groupMovement groupType;
    //...
}
```

Nuovamente la generazione degli ID è di tipo IDENTITY, si ha una nuova annotazione (Enumerated) per la gestione dell'enum.

Anche in questa entità è presente un constraint di unicità per i l campo *authorName*, per evitare Autori con nome duplicato.

Il tipo di un gruppo a cui appartiene un autore è pertanto gestito attraverso una *enum*, che allo stato attuale contiene i seguenti valori:

```
public enum groupMovement {
          CREW,ARTISTIC;
  }
```

- **DTOs**: Questo servizio oltre all'*authorDTO* presenta i DTOs per gestire 4 tipologie di richieste:

  1. AuthorCreationRequest: richiesta di creazione di un autore:

     ```
     public class AuthorCreationRequest {
         String authorName;
         String groupName;
         String groupType
         //...
     }
     ```

  2. *AuthorUpdateGroupNameRequest*: richiesta per modificare il nome del gruppo di un autore.
  3. *AuthorUpdateGroupTypeRequest*: richiesta per modificare il tipo del gruppo di un autore.
  4. *AuthorUpdateNameRequest*: richiesta per aggiornare il nome del gruppo di un autore.

  Si riporta l'esempio di una richiesta di update, ad esempio del nome del gruppo:

```
public class AuthorUpdateGroupNameRequest {
    public String groupName;
    //...
}
```

5.2.3 Flusso dei dati

Si riporta brevemente il flusso dell'informazione nel caso in cui l'utente esegua un azione di tipo Query (read), oppure Command (CUD).

- Esecuzione **Command** (e.g creazione di un author):

  1. L'utente esegue una richiesta di creazione POST

     ```
     {
         "authorName" : "Autore Di prova",
         "groupName" : "Gruppo Di prova",
         "groupType" : "CREW"

     }
     ```

  2. La richiesta viene inoltrata da RestEasy all' *AuthorCommandEndpoint* che la gestisce come segue:

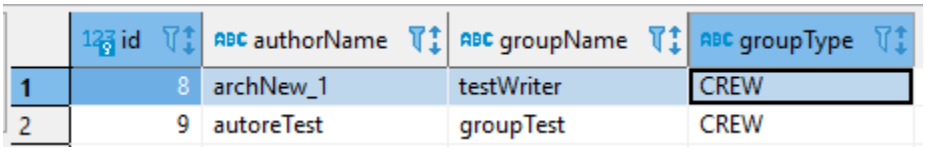
     ```
     @POST
     @Consumes(MediaType.APPLICATION_JSON)
     @Produces(MediaType.APPLICATION_JSON)
     public Response createUserFromJsonRequest(AuthorCreationCommandDTO
     ↪  request) throws Exception {
         Response response;
         try {
             Author created = authorCommandController.create(request);
             response = Response.ok(new
             ↪  AuthorMapper().AuthorEntityToDTO(created)).build();
         } catch (BadRequestException e) {
             throw e;
         }
         return response;

     }
     ```

  3. La responsabilità della gestione della richiesta viene passata all' *AuthorCommandController* il quale interagisce con il DAO per la creazione:

     ```
     public Author create(AuthorCreationRequest request) throws Exception {
             Author author = new Author();
             author.setAuthorName(request.getAuthorName());
             author.setGroupName(request.getGroupName());
             author.setGroupType(groupMovement.valueOf(request.getGroupType()));
             Author created = authorDAO.create(author);
             return created;
     }
     ```

  4. Il DAO esegue la persistenza e il controllo di consistenza, come nel caso precedente.

  La gestione delle richieste di **update** avviene in modo analogo all'UserMicroservice, l'esecuzione della richiesta precedente porta alla persistenza del nuovo autore nel database.



	id	authorName	groupName	groupType
1	8	archNew_1	testWriter	CREW
2	9	autoreTest	groupTest	CREW

Figura 12: Autore persistito nel database

Si prova quindi ad eseguire una richiesta di UPDATE (del nome dell'autore), nel seguente modo:

1. Si imposta una richiesta:
   PATCH http://localhost:8080/AuthorMicroservice/rest/AuthorMicroservice /updatename/9 con il seguente corpo:

   ```
   {
   "authorName" : "NuovoAutore"

   }
   ```

2. Il programma risponde automaticamente con un DTO relativo all'autore modificato, tuttavia si verifica che l'update abbia avuto successo, in questo caso lo si fa con una richiesta GET, di cui si riporta il risultato:

   ```
   {
           "authorName": "archNew_1",
           "groupName": "archNew_1",
           "groupType": "CREW",
           "id": 8
   },
   {
           "authorName": "NuovoAutore",
           "groupName": "NuovoAutore",
           "groupType": "CREW",
           "id": 9
   },
   ```

   In cui si nota che l'update è stato eseguito, anche in questo caso è presente una gestione delle eccezioni (es. viene inviato un ID errato, alcuni campi sono vuoti o non consistenti).

   Ad esempio nel caso in cui si richieda un author con un ID non esistente il sistema restituisce:

   ```
   404 - Autore con id X non trovato
   ```

   Allo stesso modo se si invia una richiesta di cancellazione o di un update:

   ```
   400 - Autore con id X non trovato, impossibile cancellare
   ```

   Così come nel caso in cui si invii una richiesta errata (es. nome non presente)

   ```
   400 - Valore nullo
   ```

- Esecuzione **Query** (e.g. reperire user per id)

  Supponendo di eseguire una query:

  GET http://localhost:8080/AuthorMicroservice/rest/AuthorMicroservice/9

  1. La richiesta GET viene inoltrata all' *AuthorViewEndpoint* il quale invoca l'*AuthorViewController*.
  2. L'*AuthorViewController* reperisce l'utente o restituisce un errore.
  3. I risultati vengono restituiti come DTO (un mapper converte la risposta del DAO in DTO).
  4. L'endpoint restituisce all'utente il risultato con contratto json:

```
{
    "authorName": "NuovoAutore",
    "groupName": "NuovoAutore",
    "groupType": "CREW",
    "id": 9
}
```

5.3 ArtworkMicroservice

Questo servizio è quello con l'architettura più complessa rispetto a quelli precedenti, è il servizio *core* di StreetApp in quanto permette di gestire gli artworks, e le relazioni utente-artwork, artwork-autore.

Ciò che fa questo servizio può essere riassunto nei seguenti punti:

1. Gestisce gli artworks, la gestione avviene attraverso CQRS+ES+Aggregate (La cui analisi è in subsubsection 4.0.1):
   - **Creazione:**
     (a) Un artwork viene creato attraverso una richiesta (*ArtworkReportNewRequest*) in cui si hanno i seguenti campi obbligatori:
        i. ID dell'utente che riporta l'artwork.
        ii. La latitudine.
        iii. La longitudine.
        iv. Il nome.

        Un artwork può **non** avere gli altri campi in quanto l'autore (Author_ID) potrebbe essere sconosciuto, così come lo stile.
     (b) L'artwork viene generato come nei casi precedenti a partire dalla richiesta, ma in questo caso oltre a generare un *new Artwork* si genera anche un evento.
     (c) L'evento *creazione* viene aggiunto nella lista di eventi dell'Artwork nell'event-store (*SimpleEventStore*),un artwork quindi viene persistito come una lista di Eventi.
     Tutte le richieste che *modificano lo stato* di un artwork avranno un simile percorso.
   - **Prelievo/GET**
     (a) Viene richiesto al sistema l'ID di un artwork da recuperare.
     (b) Si crea un nuovo artwork, in accordo al pattern Event Sourcing definito in [16] si implementa un metodo "rebuild" che applica ogni evento della lista dell'artwork nell'event-store al nuovo artwork.
     L'applicazione di questi eventi viene fatta in ordine, dal meno recente al più recente.
     (c) L'artwork viene convertito in un DTO.
     (d) il DTO viene inviato all'end-user finale.
   - **Update**
     (a) Viene passato al sistema l'ID di un artwork di cui fare l'update e una richiesta di update, ad esempio *ArtworkChangeNameRequest*, banalmente, una richiesta di update del *nome*.
     (b) L'artwork viene aggiornato, l'aggiornamento genera un **Evento**.
     (c) L'artwork viene aggiornato (anche) nell'event-store, si aggiunge l'ultimo evento di update nella lista di eventi dell'artwork.

- **Delete**
  La rimozione è semplice e non coinvolge particolari tecniche di Event Sourcing, è un opportuna richiesta gestita dall'endpoint (e in seguito dal controller) che dato un artwork con un certo ID ne *elimina tutti gli eventi*.

2. Comunica con gli altri microservizi (Reperisce gli utenti da UserMicroservice e gli autori da AuthorMicroservice).
   Ad esempio quando un nuovo artwork viene creato:
   (a) Nella richiesta viene inviato l'ID dell'utente che segnala un certo artwork.
   (b) La richiesta passa attraverso l'endpoint e poi al controller (come nei casi precedenti).
   (c) Il controller *usa* un oggetto di una classe Client (JAX-RS Client) e reperisce l'intero User con quell'ID.
   (d) L'artwork viene persistito nell'event-store, con il riferimento all'user.

   Questo serve perchè in ArtworkMicroservice è opportuno avere lo stato "reale" di autori e utenti per la creazione di report (es. report storici event-oriented) e lo storing di eventi. Queste classi *Client* hanno il ruolo di richiedere i dati dagli opportuni servizi.

5.3.1 Metodi Implementati

L'ArtworkMicroservice espone i seguenti metodi:

- GET *http://localhost:8080/ArtworkMicroservice/rest/ArtworkMicroservice/{id}*: metodo (get) per reperire un artwork dato l'ID.
- GET *http://localhost:8080/ArtworkMicroservice/rest/ArtworkMicroservice/*: metodo che restituisce tutti gli artworks.
- GET *http://localhost:8080/ArtworkMicroservice/rest/ArtworkMicroservice/gethistory/{id}*: metodo che restituisce tutti gli eventi di un artwork.
- POST *http://localhost:8080/ArtworkMicroservice/rest/ArtworkMicroservice/*: metodo per la creazione di un Artwork, riceve una richiesta di tipo *ArtworkReportNewRequest*, in cui i campi citati nella sezione precedente sono *obbligatori* e crea un nuovo artwork.

```
public class ArtworkReportNewRequest {
    private String name;
    private String style;
    private String type;
    private long latitude;
    private long longitude;
    private int reportingUserID;
    private int artworkCreatorID;
}
```

- DELETE *http://localhost:8080/ArtworkMicroservice/rest/ArtworkMicroservice/{id}*: metodo per rimuovere un artwork dato l'ID.
- Varie richieste di update (parziali) :

1. PATCH *http://localhost:8080/ArtworkMicroservice/rest/ArtworkMicroservice/updatestyle/{id}*: metodo per fare update dello stile di un artwork.
2. PATCH *http://localhost:8080/ArtworkMicroservice/rest/ArtworkMicroservice/updatetype/{id}*: metodo per fare update del tipo di un artwork.
3. PATCH *http://localhost:8080/ArtworkMicroservice/rest/ArtworkMicroservice/updatename/{id}*: metodo per fare update del nome di un artwork.
4. PATCH *http://localhost:8080/ArtworkMicroservice/rest/ArtworkMicroservice/updateauthor/{id}*: metodo per fare update dell'autore nome di un artwork.

5.3.2 Architettura

L'ArtworkMicroservice ha l'architettura più complessa se paragonato agli altri servizi, verrà quindi descritta nel dettaglio.

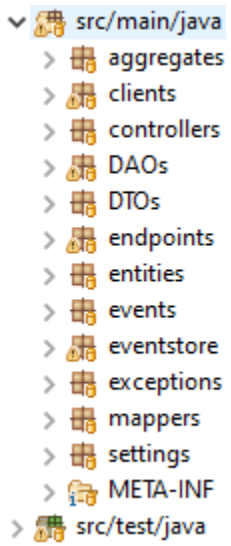


Figura 13: Architettura Artwork Service

- **aggregates :** Questo progetto implementa rigorosamente il pattern Aggregate descritto da Richardson [18], il cui funzionamento teorico è descritto in 4.0.1.3

  Questo package contiene due classi *fondamentali*: *AggregateRoot, ArtworkAggregate.*

  - ***AggregateRoot***: E' una classe *astratta* che rappresenta la radice dell'aggregato, gestisce l'ID (implementato attraverso il framework UUID), mantiene una lista di eventi relativi all'aggregato e un valore (int) che rappresenta la versione.

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

    Ogni aggregato dovrà *estendere* aggregateRoot, questa classe è vitale per il funzionamento di tutta l'architettura, in quanto tiene traccia dell'ID e gestisce gli eventi nel seguente modo:

1. Gli eventi sono salvati come una lista.
2. Un metodo *generateEvent(Event e)* applica l'evento che viene passato come argomento, e se l'aggregato si trova ad una versione precedente di quella dell'evento, applica l'evento ed incrementa il valore della versione (che di default è 0).
   Se viene passato un evento "vecchio" allora non viene applicato.
3. Un metodo *rebuildAggregateStatus(List < Event > history)* realizza la ricostruzione dell'aggregato ad eventi descritta da Richardson, si applicano tutti gli eventi della lista in modo ordinato (dal meno recente al più recente) partendo da un aggregato *nuovo*.

```
public final void rebuildAggregateStatus(List<Event> history) {
        history.sort(new EventComparator()); // Ordino gli eventi per
        ↪  versione
        for (Event e : history) { // Li applico
                applyChange(e, false);
        }
        this.events = history;
        this.version = history.get(history.size() - 1).eventNumber;
}
```

4. L'applicazione degli eventi avviene nel seguente modo, la classe concreta (cioè l'aggregato concreto) esporrà dei metodi *apply* per ogni evento, in questo caso (per un artwork) si avranno i seguenti metodi apply:

```
    public void apply(ArtworkNameChangedEvent event) {
                this.name = event.newname;
        }

        public void apply(ArtworkStyleChangedEvent event) {
                this.style = event.newstyle;
        }
```

   Dalla classe astratta (root) alla classe concreta dato un certo evento *e* si risale all'opportuno metodo apply attraverso un metodo manuale, che permette di evitare la reflection, questo metodo consiste in 2 parti:

   (a) Si dichiarano gli eventi come metodi astratti nella superclasse astratta AggregateRoot:

```
public abstract void apply(ArtworkNameChangedEvent event);
public abstract void apply(ArtworkStyleChangedEvent event);
public abstract void apply(ArtworkTypeChangedEvent event);
//...
```

   (b) Si definisce un metodo *invokeApplyMethodNoReflection(Event e)* per applicare gli eventi dalla root astratta all'aggregate concreto:

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

5. Nella classe AggregateRoot si hanno anche getters/setters per la gestione degli eventi.

– ***ArtworkAggregate***: Aggregato che implementa l'artwork, ovviamente estende aggregateroot, espone tutti i metodi apply ed ha un costruttore con i 4 elementi obbligatori per generare un artwork.

Il costruttore è un primo metodo in cui si *genera* un evento.

```
public ArtworkAggregate(User reportingUser, long latitude, long longitude,
↪   String name) {
    super(UUID.randomUUID());
    this.reportingUser = reportingUser;
    this.latitude = latitude;
    this.longitude = longitude;
    this.name = name;
    //Generazione dell'evento:
    generateEvent(new ArtworkCreatedEvent(super.getId(),
                  reportingUser, latitude, longitude, name));
}
```

Una cosa importante che si nota è che gli ID degli artworks non sono *int* come negli altri casi ma sono degli *UUID*, questo è dovuto a due fattori principali:

1. L'event-store è in-memory, quindi non c'è un meccanismo di gestione degli ID (equivalente degli auto-increment del DB) pertanto con UUID.RandomUUID() si possono generare facilmente UUID freschi e casuali.
2. Alcuni elementi, come gli *eventi* hanno la necessità di ricevere subito alla loro creazione un ID univico universale.

Tutti i metodi di ArtworkAggregate che modificano lo stato *generano un evento*

```
//Eventi
public void changeArtStyle(ArtStyle style) {
        this.style = style;
        ArtworkStyleChangedEvent evt = new
        ↪  ArtworkStyleChangedEvent(super.getId(), style);
        generateEvent(evt);
}

public void changeArtType(ArtType type) {
        this.type = type;
        generateEvent(new ArtworkTypeChangedEvent(super.getId(), type));
}

public void changeName(String newName) {
        this.name = newName;
        generateEvent(new ArtworkNameChangedEvent(super.getId(),
        ↪  newName));
}

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

}
```

Anche la cancellazione produce un evento, che funge da placeholder per segnare che l'artwork è stato rimosso (e.g cancellato).

- **clients :** Questo package contiene dei clients (JAX-RS Client), sono 2, *UserClient* e *AuthorClient*, un esempio è il metodo che permette di reperire (es. un user) dall'User-Microservice conoscendo l'ID.

```
public  User getUserFromRestApiById(int id) throws Exception {

    String target =
    ↪  "http://localhost:8080/UserMicroservice/rest/UserMicroservice/{id}";

    Client client = ClientBuilder.newClient();
    WebTarget myResource = client.target(target).resolveTemplate("id", id);
    UserDTO response =
    ↪  myResource.request(MediaType.APPLICATION_JSON).get(UserDTO.class);

    UserMapper u = new UserMapper();
    User retrieved = u.DTOToUser(response);
    client.close();
    return retrieved;
}
```

Qui si nota l'interazione tra Mapper, DTO e Entities.

- **endpoints :** Come negli altri microservizi, si hanno due endpoints, uno per le operazioni R (READ) e uno per le CUD (Create, Update,Delete), per brevità ometto la descrizione nel dettaglio dei suddetti endpoints, si riporta il metodo di creazione del command endpoint e di visualizzazione del view endpoint.

```
@POST
@Produces(MediaType.APPLICATION_JSON)
public Response create(ArtworkReportNewCommandDTO request) throws
↪  BadRequestException {
        Response response;
        try {
                ArtworkAggregate created = service.createArtwork(request);
                response = Response.ok(new
                ↪  ArtworkMapper().ArtworkEntityToDTO(created)).build();
        } catch (Exception e) {
                throw new BadRequestException("Impossibile creare l'artwork con
                ↪  la richiesta effettuata");
        }
        return response;
}

@Path("/{id}")
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response findArtworkByID(@PathParam("id") String id) throws Exception {
        Response response;
        UUID uuid = UUID.fromString(id);
        try {
                ArtworkDTO resultDTO = service.getArtworkByUUID(uuid);
                response = Response.ok(resultDTO).build();
        } catch (ResourceNotFoundException e) {
                throw e;
        }
        return response;

}
```

- **DAOs :** Il DAO è analogo agli altri, ma le operazioni vengono invocate *sull'event-store* anzichè sul database attraverso JPA.

  A titolo di esempio per sottolineare l'uniformità dell'interfaccia dell'event-store si riporta il metodo save (Che persiste nell'event-store).

```
public void save(ArtworkAggregate a) throws Exception {
        eventStore.save(a.getId(), a.getEvents());
}
```

  Il metodo update è :

```
public ArtworkAggregate update(ArtworkAggregate newArtwork, UUID id) throws
↪  Exception {
        eventStore.update(newArtwork, id);
        ArtworkAggregate updatedArtwork = new ArtworkAggregate(id);
        updatedArtwork.rebuildAggregateStatus(eventStore.load(id));
        return updatedArtwork;

}
```

Nel metodo sopracitato si nota la presenza del metodo per ricostruire un artwork a partire dai suoi eventi.

- **DTOs :** Il package DTO è complesso rispetto agli altri servizi, contiene un totale di 9 classi, divise come segue:

  1. DTOs relativi alle **entità**: in questa categoria rientrano artworkDTO,UserDTO e authorDTO che hanno un ruolo anaologo agli altri DTO degli altri microservizi.
  2. DTOs relativi alle **richieste**: si hanno i seguenti DTO relativi alle richieste che possono essere inviate al servizio:
     (a) *ArtworkChangeAuthorCommandDTO*: Implementa la richiesta di cambio di autore di un artwork.
     (b) *ArtworkChangeNameCommandDTO*:Implementa la richiesta di cambio del nome di un artwork.
     (c) *ArtworkChangeStyleCommandDTO*: Implementa la richiesta di cambio dello stile di un artwork.
     (d) *ArtworkChangeTypeCommandDTO*: Implementa la richiesta di cambio del tipo di un artwork.
     (e) *ArtworkReportnewCommandDTO*: Implementa la richiesta di creazione di un nuovo artwork.
  3. DTOs relativi agli **eventi**, questo contiene un EventDTO, si utilizza per il metodo GetHistory che permette di ricevere la storia di un artwork come lista di eventi, pertanto riporta le informazioni di un evento.

- **entities :** Tra le entities di ArtworkMicroservice si possono trovare le entità Author e User degli altri servizi e due *enum*: ArtStyle e ArtType per la descrizione dello stile e del tipo di un artwork.

  Stile e tipo quindi, sono gestiti con delle enum, in breve:

```
public enum ArtStyle {
          HYPERREALISTIC, COMIC;
}
public enum ArtType {
          MURALES, STICKER;
}
```

- **events :** Questo è uno dei maggiori cambiamenti di ArtworkMicroservice, si implementano gli *eventi*, in questo package si hanno:

  1. Una classe astratta "***Event***" che ha i seguenti campi:

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

ovvero:

- Un UUID di un aggregate (in questo caso ArtworkAggregate) a cui si riferisce.
- Un intero *eventNumber* che corrisponde alla versione dell'aggregato.
  Questo serve per allineare i cambiamenti dell'artwork e degli eventi, infatti in metodi come *rebuildAggregateStatus* dell'artwork Aggregate gli eventi devono essere applicati *in ordine (ovviamente)* pertanto questo campo fornisce l'ordinamento degli eventi.

2. Una classe ***EventComparator***: questa classe è un *comparator*, si usa per ordinare gli eventi in ordine crescente nel metodo *rebuildAggregateStatus*, come segue:

```
public final void rebuildAggregateStatus(List<Event> history) {
    history.sort(new EventComparator()); // Ordino gli eventi per
    ↪ versione
    // ...

}
```

La classe comparator è un semplice comparatore di versioni:

```
public class EventComparator implements Comparator<Event> {
        @Override
        public int compare(Event o1, Event o2) {
                return o1.eventNumber - o2.eventNumber;
        }
}
```

3. ***Eventi Concreti*** si hanno vari eventi per un artwork, questi sono AuthorChanged, NameChanged etc. (per tutti i possibili cambiamenti di stato).
   Un evento che si può riportare a titolo d'esempio è il seguente:

```
public class ArtworkStyleChangedEvent extends Event {
        public ArtStyle newstyle;

        public ArtworkStyleChangedEvent(UUID uuid, ArtStyle newname) {
                super();
                super.aggregateId = uuid;
                this.newstyle = newname;
        }
}
```

- **eventstore :** Questo package è fondamentale per la gestione degli eventi, ha una sola classe "SimpleEventStore" con 2 features principali:

1. Gestisce gli eventi, dato un aggregato gestisce una lista di eventi a lui relativi.
2. E' una classe singleton.
3. Ha una gestione di base di concorrenza.

I componenti fondamentali sono:

1. Gestione eventi e istanza (singleton):

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

//Singleton
public static SimpleEventStore getInstance() {
        return instance;
}
```

2. Metodi CRUD su eventi, questi metodi lanciano delle specifiche eccezioni user-defined che vengono poi propagate all'utente finale, si riporta la load e la save:

```
// Si salvano tutti gli eventi di un aggregato
// Se un aggregato è allo stato precedente, si sovrascrivono.
public void save(UUID uuid, List<Event> newEvents) throws Exception {
        if (newEvents==null) {
                throw new BadRequestException("Richiesta non valida");
        }
        events.put(uuid, newEvents);
}

// Si caricano gli eventi di un aggregato
public List<Event> load(UUID aggregateId) throws ResourceNotFoundException {
        List<Event> aggregateEvents = events.get(aggregateId);
        if(aggregateEvents==null){
                throw new ResourceNotFoundException("ID non valido,
                                non corrisponde a nessun artwork");
        }
        return new ArrayList<>(aggregateEvents);
}
```

- **mappers :** Il mapper ha un funzionamento analogo agli altri mappers analizzati precedentemente, si è aggiunto un EventMapper per convertire gli Events in EventDTO.

  Si hanno anche i mappers relativi alla gestione delle eccezioni, nello specifico si implementano le eccezioni ResourceNotFoundException e BadRequestException, nel package mappers si avranno i relativi *ExceptionMapper* Jax-Rs.

  1. *BadRequestExceptionMapper.*
  2. *ResourceNotFoundExceptionMapper.*

  Si riporta a titolo di esempio il mapper per un eccezione di richiesta malformata:

```
@Provider
public class BadRequestExceptionMapper implements
↪  ExceptionMapper<BadRequestException> {
        @Override
        public Response toResponse(BadRequestException exception) {
                return Response.status(400).entity(exception.getMessage())
                .type(MediaType.TEXT_PLAIN).build();
        }

}
```

- **controllers :** I controller come negli altri progetti sono due, corrispondenti agli endpoints, si avrà quindi un *ArtworkCommandController* ed un *ArtworkViewController*, una grossa differenza si nota nelle funzioni di update in cui si vanno ad aggiungere eventi e di get in cui si ricostruisce un aggregato. Si riportano due esempi, l'esecuzione di un command (Update) e di una query.

  - **Update**: Nel metodo per modificare lo stile di un artwork nella classe controller si va a ricostruire l'aggregato secondo il pattern EventSourcing, si fa la modifica (Che inserirà un nuovo evento) ed infine si persiste.

    ```
    public ArtworkAggregate changeArtworkStyle(ArtworkChangeStyleCommandDTO
    ↪  request, UUID fromString) throws Exception {
        if (request.style == null || request.style.equals("")) {
                throw new BadRequestException("Valore stile nullo");
        }
        if (!Stream.of(ArtStyle.values())
        .anyMatch(v->v.name().equals(request.getStyle()))){
                throw new BadRequestException("Valore dello stile non
                ↪  presente, valori presenti: " +
                ↪  ArtStyle.values().toString());
        }
        ArtworkAggregate a = new ArtworkAggregate(fromString);
        a.rebuildAggregateStatus(dao.load(fromString).getEvents());
        a.changeArtStyle(ArtStyle.valueOf(request.getStyle()));
        ArtworkAggregate updated = dao.update(a, fromString);
        return updated;
    }
    ```

    Si nota il flusso modificato per gestire gli eventi.

  - **GET** Ad esempio il metodo per ricevere un artwork dato il suo ID ha il seguente funzionamento

    ```
    public ArtworkDTO getArtworkByUUID(UUID id) throws Exception {
            ArtworkMapper mapper = new ArtworkMapper();
            ArtworkAggregate a =  dao.load(id);
            return mapper.ArtworkEntityToDTO(a);

    }
    ```

    In cui nel metodo load del DAO si esegue il rebuild dell'aggregato.

L'utente finale si interfaccerà con l'endpoint di ArtworkMicroservice, salvo per la registrazione (in cui subentra l'userMicroservice) oppure inserimento di un nuovo autore (AuthorMicroservice).

6 Implementazione (Basi di dati)

6.0.1 Basi di dati

I tre servizi hanno i seguenti metodi di memorizzazione dei dati:

1. AuthorMicroservice → database Mysql 8.0.
2. UserMicroservice → database Mysql 8.0
3. ArtworkMicroservice → event-store in-memory costruito "ad-hoc" (il dettaglio sul funzionamento verrà riportato nella sezione implementazione.

Nelle due sottosezioni seguenti si analizzano le basi di dati di UserMicroservice e AuthorMicroservice, si omette la descrizione di quella di ArtworkMicroservice in quanto l'implementazione del "SimpleEventStore" è presente in item 5.3.1.

6.0.2 Base di dati UserMicroservice

Il DBMS a cui si appoggia UserMicroservice, ovvero "users_db" è MySQL 8.0, è presente soltanto la tabella "user" con 2 campi, un campo "id" (Integer autoincrement primary key not null) ed un campo "username" (varchar(255)).

L'utente con cui si accede al database allo stato attuale è l'utente *root*.

6.0.3 Base di dati AuthorMicroservice

Il DBMS a cui si appoggia AuthorMicroservice, ovvero "authors_db" è MySQL 8.0, è presente soltanto la tabella "author" con 4 campi:

1. Un campo id int autoincrement not null primary key.
2. Un campo authorName (varchar(255)).
3. un campo groupName (varchar(255)).
4. un campo groupType (varchar(255)).

L'utente con cui si accede al database allo stato attuale è l'utente *root*.

7 Analisi di funzionamento

Per mostrare il funzionamento di StreetApp si analizza un esempio di caso reale:

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 sovrascritto *interamente*.
4. L'artwork viene cancellato.

Per semplicità si suppone anche che un utente (id=23, nome= "NuovoUsername") e un autore (id=9, nome = "AutoreTest"), siano già stati creati.

L'utente individua un nuovo artwork, decide si segnalarlo sul sistema, supponendo che l'artwork abbia le seguenti caratteristiche:

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

Esegue quindi una richiesta di creazione (su ArtworkMicroservice), la richiesta avrà il corpo come nel codice precedente e sarà del tipo:
POST http://localhost:8080/AuthorMicroservice/rest/AuthorMicroservice/

Questa richiesta restituisce come *Response* il DTO dell'artwork appena creato, incluso il suo ID, in questo caso "27d1fa20-9240-49ec-9dee-87de53e27a3e"

Eseguendo una *GET* del tipo:
GET http://localhost:8080/AuthorMicroservice/rest/AuthorMicroservice/27d1fa20-9240-49ec-9dee-87de53e27a3e

si restituiscono le informazioni relative all'artwork appena persitito:

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

Si suppone poi che nel corso del tempo questo artwork venga modificato, nello specifico subirà i seguenti eventi [1]:

[1] Un evento non potrà *mai* modificare i dati core come latitudine, longitudine e l'utente che lo ha riportato per primo.

1. Un utente trova un nome più appropriato, ad esempio viene modificato il nome.

   PATCH http://localhost:8080/ArtworkMicroservice/rest/ArtworkMicroservice /updatename/27d1fa20-9240-49ec-9dee-87de53e27a3e
   Con corpo:

   ```
   {
       "name" : "Graffito di via Y"

   }
   ```

2. Un utente può assegnare all'artwork uno stile artistico più appropriato, es. STICKER piuttosto che MURALES.
   PATCH http://localhost:8080/ArtworkMicroservice/rest/ArtworkMicroservice /updatetype/27d1fa20-9240-49ec-9dee-87de53e27a3e
   Con corpo:

   ```
   {
       "type" : "STICKER"

   }
   ```

Ognuno di questi eventi nel momento della sua invocazione restituisce una Response contenente il DTO dell'artwork modificato, nel caso in cui si sollevino delle eccezioni, queste verrano inviate come risposta (con opportuno codice) all'utente finale.

A titolo di esempio, una richiesta di creazione senza campi obbligatori, come latitudine e longitudine genererà un eccezione di tipo:

```
400 - Impossibile creare l'artwork con la richiesta effettuata
```

In totale questi eventi portano il seguente cambiamento di stato (eseguendo la richiesta GET con apposito ID)

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

Un cambiamento tuttavia può essere anche più **drastico**, un artwork può essere sovrascritto:

1. L'autore 10 sovrascrive l'artwork (per semplicità lo stile rimane lo stesso), l'artwork cambierà di nome, l'autore (e in linea teorica la risorsa .jpg).
   Cambio di autore:
   PATCH http://localhost:8080/ArtworkMicroservice/rest/ArtworkMicroservice /updateauthor/27d1fa20-9240-49ec-9dee-87de53e27a3e
   con corpo:

```
{
        "artworkCreator" : 10

}
```

Cambio di nome:
PATCH http://localhost:8080/ArtworkMicroservice/rest/ArtworkMicroservice /updatename/27d1fa20-9240-49ec-9dee-87de53e27a3e
con corpo:

```
{
        "name" : "Graffito Sovrascritto"

}
```

Risultato (GET)


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


Si può notare che l'artwork è stato correttamente aggiornato.

L'artwork in questione avrà una lista di eventi pari a 5 e attraverso il pattern *event sourcing* sarà possibile non solo ricostruirlo (come viene fatto nella GET) ma anche *ricostruire stati precedenti* e accedere a tutto lo storico degli eventi.

La granularità delle richieste di update è utile per storicizzare cambiamenti di stato che riguardano anche il singolo campo.

Il sistema allo stato attuale è in grado di inviare all'utente attraverso una richiesta:

GET http://localhost:8080/ArtworkMicroservice/rest/ArtworkMicroservice/gethistory/$\{ID\}$

La storia di tutti gli eventi di un aggregato (sottoforma di $List < EventDTO >$), con il relativo numero di evento (versione), si riporta un esempio (slegato dall'esempio precedente):

```
{
        "eventName": "class events.ArtworkCreatedEvent",
        "version": 1
},
{
        "eventName": "class events.ArtworkStyleChangedEvent",
        "version": 2
},
{
        "eventName": "class events.ArtworkTypeChangedEvent",
        "version": 3
},
{
        "eventName": "class events.ArtworkAuthorChangedEvent",
        "version": 4
},
{
        "eventName": "class events.ArtworkNameChangedEvent",
        "version": 5
}
```

La cancellazione è un particolare evento che viene persistito:

```
{
    "eventName": "class events.ArtworkRemovedEvent",
    "version": x
}
```

8 Testing

Nel sistema sono stati implementati vari **test** attraverso *Junit 4.12* e *Mockito 4.2.0*.

1. Ogni microservizio implementa test per il servizio di gestione delle queries (View controller) e per gli endpoints (command endpoint).

   Questi due classi per ogni microservizio costituiscono il "nucleo funzionale" in quanto gestiscono le richieste inoltrare dal controller ed interagiscono con i DAOs.

2. Alcuni microservizi implementano test specifici, ad esempio l'ArtworkMicroservice implementa un test per l'event-store in cui si vanno a verificare le operazioni CRUD.

Verranno quindi presentati i test divisi in sezioni corrispondenti ad i vari servizi, e sottosezioni corrispondenti agli specifici tests.

8.1 Test in UserMicroservice

Si eseguono test dell'user *command service* e user *view service*.

8.1.1 Test Command Endpoint

Si eseguono i test per le operazioni CUD:

```
@Test
public void testDelete() throws Exception {
        User u = new User();
        u.setUsername("Test_Username");
        //Utente viene rimosso
        when(userDAO.delete(Mockito.any(Integer.class))).thenReturn(u);
        User deleted = commandService.delete(u.getId());
        assertEquals(deleted, u);
}

@Test
public void testUpdate() throws Exception {
        User u = new User();
        u.setUsername("user");
        //Update
        u.setUsername("user2");
        UserUsernameUpdateRequest request = new UserUsernameUpdateRequest();
        request.setUsername("user2");
        when(userDAO.update(u.getId(),u)).thenReturn(u);
        User updated = commandService.update(u.getId(), request);
        assertEquals(updated, u);
}

@Test
public void testCreation() throws Exception {
        User user = new User();
        user.setUsername("test_username");
        UserCreationRequest request = new UserCreationRequest();
        request.setUsername(user.getUsername());
```

```
        when(userDAO.create(Mockito.any(User.class))).thenReturn(user);
        User created = commandService.create(request);
        assertEquals(user, created);
}
```

8.1.2 Testing View Controller

In questa sezione si esegue soltanto un test per il metodo che permette di reperire un utente dato il suo id.

```
@Test
public void testFindById() throws Exception {
        User a = new User();
        a.setUsername("Test");
        UserMapper mapper = new UserMapper();
        UserDTO a_dto = mapper.UserToDto(a);
        when(userDAO.findById(Mockito.any(Integer.class))).thenReturn(a);
        UserDTO service_result = viewService.findById(123);
        Assert.assertEquals(a_dto,service_result);
}
```

Entrambi i test danno esito positivo.

8.2 Test in AuthorMicroservice

Si eseguono test dell'*author command service* e *author view service*

8.2.1 Test Command Endpoint

Si eseguono i test per le operazioni CUD, per brevità si riportano soltanto i metodi delete e creation.

```
@Test
public void testDelete() throws Exception {
        Author testAuthor = new Author();
        testAuthor.setAuthorName("Autore di test");
        testAuthor.setGroupName("Gruppo di test");
        testAuthor.setGroupType(groupMovement.ARTISTIC);
        //Utente viene rimosso
        when(dao.delete(Mockito.any(Integer.class))).thenReturn(testAuthor);
        Author deleted = commandService.delete(testAuthor.getId());
        assertEquals(deleted, testAuthor);
}

@Test
public void testCreation() throws Exception {
        Author testAuthor = new Author();
        testAuthor.setAuthorName("Autore di test");
        testAuthor.setGroupName("Gruppo di test");
        testAuthor.setGroupType(groupMovement.ARTISTIC);
        when(dao.create(Mockito.any(Author.class))).thenReturn(testAuthor);
        AuthorCreationRequest creationRequest = new AuthorCreationRequest();
        creationRequest.setAuthorName(testAuthor.getAuthorName());
```

```
        creationRequest.setGroupName(testAuthor.getGroupName());
        creationRequest.setGroupType(testAuthor.getGroupType().toString());
        Author created = commandService.create(creationRequest);
        assertEquals(created, testAuthor);
}
```

8.2.2 Testing View Controller

Si riporta il test del metodo *findbyid()*.

```
@Test
public void testFindById() throws Exception {
        Author testAuthor = new Author();
        testAuthor.setAuthorName("Autore di test");
        testAuthor.setGroupName("Gruppo di test");
        testAuthor.setGroupType(groupMovement.ARTISTIC);
        AuthorMapper mapper = new AuthorMapper();
        AuthorDTO a_dto = mapper.AuthorEntityToDTO(testAuthor);
        when(authorDAO.findByID(Mockito.any(Integer.class))).thenReturn(testAuthor);
        AuthorDTO service_result = viewService.findById(123);
        Assert.assertEquals(a_dto,service_result);
}
```

Entrambi i test danno esito positivo.

8.3 Test in ArtworkMicroservice

In ArtworkMicroservice si hanno i test per l' *artwork command service*, per l'*artwork view service* ed infine per l'*event-store in-memory*.

8.3.1 Test Command Controller

Si sono implementati i test per le operazioni principali, ad esempio per la creazione e eliminazione si hanno:

```
@Test
public void testArtworkCreation() throws Exception {
    // Si crea un aggregato di prova.
    ArtworkAggregate artwork = getTestingArtwork();

    // Si genera una richiesta di creazione a partire dall'artwork
    ArtworkReportNewRequest creationRequest = new ArtworkReportNewRequest();
    creationRequest.setLatitude(artwork.getLat());
    creationRequest.setLongitude((artwork.getLongitude()));
    creationRequest.setName(artwork.getName());
    creationRequest.setReportingUserID(artwork.getReportingUser().getId());
    creationRequest.setStyle(artwork.getStyle().toString());
    creationRequest.setType(artwork.getType().toString());
    creationRequest.setArtworkCreatorID(artwork.getArtworkCreator().getId());

    // Mocking
    when(userClient.getUserFromRestApiById(any(Integer.class)))
                                        .thenReturn(artwork.getReportingUser());
```

```
    when(authorClient.getAuthorFromRestApiById(any(Integer.class)))
                                        .thenReturn(artwork.getArtworkCreator());
    when(dao.save(any(ArtworkAggregate.class))).thenReturn(artwork);

    // Si esegue la richiesta
    ArtworkAggregate created = commandService.createArtwork(creationRequest);
    assertEquals(artwork, created);

}

@Test
public void testArtworkDelete() throws Exception {
        //Si crea un artwork di test
        ArtworkAggregate artwork = getTestingArtwork();
        artwork.removeArtwork();
        when(dao.load(artwork.getId())).thenReturn(artwork);
        when(dao.update(any(ArtworkAggregate.class), any(UUID.class))).thenReturn(artwork);
        //Gestione della cancellazione
        ArtworkAggregate deleted = commandService.deleteArtwork(artwork.getId());
        assertEquals(artwork, deleted);
}
```

Mentre per le varie richieste di update si hanno dei singoli test, a titolo di esempio ne viene riportato uno:

```
@Test
public void TestChangeAuthor() throws Exception {
    // Si crea un aggregato di prova.
    ArtworkAggregate artwork = getTestingArtwork();

    // Si crea una richiesta di update dell'autore
    Author newAuthor = new Author();
    artwork.changeAuthor(newAuthor);
    ArtworkChangeAuthorRequest authorUpdateRequest = new ArtworkChangeAuthorRequest();
    authorUpdateRequest.artworkCreator = 1;

    // Mocking
    when(dao.update(any(ArtworkAggregate.class), any(UUID.class))).thenReturn(artwork);
    when(dao.load(any(UUID.class))).thenReturn(artwork);

    // Si esegue la richiesta
    ArtworkAggregate updated = commandService
                .changeArtworkAuthor(authorUpdateRequest, artwork.getId());
    assertEquals(artwork.getArtworkCreator().getId(), updated.getArtworkCreator().getId());

}
```

In cui si testa una richiesta di tipo ”cambio di autore”.

8.3.2 Testing View Endpoint

Per il servizio di gestione delle queries, si riporta il metodo per testare la GET by id:

```
@Test
public void testFindById() throws Exception {
        ArtworkAggregate artwork = getTestingArtwork();
        UUID target_uuid = UUID.randomUUID();
        ArtworkMapper mapper = new ArtworkMapper();
        ArtworkDTO a_dto = mapper.ArtworkEntityToDTO(artwork);
        when(artworkDAO.load(Mockito.any(UUID.class))).thenReturn(artwork);
        ArtworkDTO service_result = viewService.getArtworkByUUID(target_uuid);
        Assert.assertEquals(a_dto,service_result);
}
```

Dove il metodo *getTestingArtwork()* crea un artwork con i vari campi a titolo di test (ed evitare codice duplicato).

8.3.3 Testing SimpleEventStore

E' stato eseguito il testing (JUnit 4.12) dell' in-memory event-store, questo test riguarda le operazioni CRUD sugli eventi, si riportano i test, iniziando dalla creazione e caricamento:

```
        @Test
        public void testSaveAndLoad() throws Exception {
                // si salva l'aggregato
                eventStore.save(testArtwork.getId(), testArtwork.getEvents());
                ArtworkAggregate retrieved = new ArtworkAggregate(testArtwork.getId());
                retrieved.rebuildAggregateStatus(eventStore.load(testArtwork.getId()));
                Assert.assertEquals(testArtwork, retrieved);
        }
```

Segue upload e cancellazione:

```
    @Test
        public void testUpdate() throws Exception {
                eventStore.save(testArtwork.getId(), testArtwork.getEvents());
                testArtwork.changeName("Update test");
                testArtwork.changeArtStyle(ArtStyle.COMIC);
                testArtwork.changeArtType(ArtType.MURALES);
                testArtwork.changeAuthor(new Author());
                eventStore.update(testArtwork, testArtwork.getId());
                ArtworkAggregate loadedAggregate = new ArtworkAggregate(testArtwork.getId());
                loadedAggregate.rebuildAggregateStatus(eventStore.load(testArtwork.getId()));
                Assert.assertEquals(loadedAggregate, testArtwork);
        }

        @Test(expected = NullPointerException.class)
        public void testDelete() throws Exception {
                eventStore.save(testArtwork.getId(), testArtwork.getEvents());
                eventStore.delete(testArtwork.getId());
                eventStore.load(testArtwork.getId());
        }
```

9 Osservazioni e Conclusioni

In questo progetto ho avuto modo di approfondire l'argomento delle architetture software, a me quasi del tutto nuovo.

Il progetto ha riguardato l' analisi nel dettaglio di architetture software orientate ai *microservizi*, pattern rilevanti in questo contesto: CQRS, EventSourcing, Aggregate.

Prima di implementare *Street App* sono partito analizzando il domain model, identificando i bounded contexts i potenziali microservizi secondo il metodo DDD.

Si sono quindi implementati 3 microservizi, in cui se ne distingue uno "principale": ArtworkMicroservice con cui si interfaccia l'utente finale, in cui si ha l'implementazione di EventSourcing ed è interamente *orientato agli eventi*.

Un osservazione importante riguarda l'architettura utilizzata. Un sistema orientato a **microservizi** si è rivelato essere estremamente *flessibile*, in grado di migliorare il processo di creazione e mantenimento di un applicazione e capace di gestire casi d'uso molto diversi.

Rimangono tuttavia delle problematiche da tenere in considerazione [18] :

- **Latenza**: Un architettura che massimizza la distribuzione sulla rete può portare problemi di latenza.
- **Comunicazione sincrona:** La comunicazione generalmente è sincrona, ad esempio un servizio A richiede un entità ad un servizio B, questo può creare problemi nella gestione dei fallimenti, ad esempio il servizio A non risponde.
- **Consistenza dei dati:** Questo è forse il problema più rilevante, se si aggiornano i dati relativi ad una certa entità in un servizio, questi cambiamenti dovranno essere *propagati* a tutti gli altri servizi che "conoscono" quell'entità.
- **Decomposizione in microservizi:** In alcuni domini può essere difficile individuare una potenziale divisione in microservizi ad esempio sistemi che contengono classi god [18].

Eseguendo una breve analisi sui pattern, **CQRS** si è dimostrato *fondamentale* in tutti i microservizi per:

1. Dare un ordine logico ai vari componenti funzionali: CQRS permette di dividere le operazioni CUD dalle read.
2. Rendere scalabile il sistema: CQRS permette di ridurre il carico di lavoro su un unico *endpoint* permettendo "viste" sui dati che possono essere modificate in corso d'opera.
3. Separare le responsabilità: attraverso CQRS è più facile identificare e risolvere problemi, essendo i due flussi disaccoppiati.

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

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

Il pattern **Aggregate** diviene fondamentale a mio avviso per domini complessi indipendentemente dall'architettura in analisi, questo perché:

1. Permette di gestire più facilmente cluster di oggetti, per via della presenza dell' *aggregate root*.

   Questo oltre ad un miglioramento della gestione a livello logico porta anche un risvolto positivo sulla *security* del programma.

2. Dal punto di vista della security, operazioni *critiche* che interessano tutto il cluster possono essere svolte proprio sulla root, minimizzando la superficie di rischio e quindi di potenziale fallimento.

CQRS e Aggregate sono quindi ottimi per sicurezza, scalabilità e gestione a livello logico di un architettura software.

Il pattern **Event Sourcing** invece può non essere conveniente se non in casi *estremamente ristretti*, questo perchè presenta alcuni problemi:

1. La procedura di "ricostruzione" di un aggregato dati i suoi eventi ha un costo *lineare* rispetto al numero di eventi stessi.

   Questo può essere indifferente in contesti come l'app. StreetApp in cui un aggiornamento di un artwork si può supporre essere poco frequente.

   In contesti molto dinamici tuttavia, ad esempio un servizio bancario, ricostruire lo stato di un cliente attraverso i suoi eventi può essere estremamente inefficiente.

2. Il numero di eventi tende a crescere rapidamente, non solo al crescere della complessità del dominio, ma alla necessità di gestire eventi *combinati* rispetto ad eventi semplici ma che devono essere a loro volta definiti come 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.

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.

   Un implementazione che sembra essere stata ricevuta positivamente è Axon (axoniq.io).

Limiterei quindi l'utilizzo di questo pattern alle seguenti **circostanze**:

1. Il dominio di applicazione è semplice e relativamente statico.
2. Gli eventi sono pochi, e la frequenza di "generazione" degli eventi è molto ridotta.
3. I sistemi non abbiano carico di lavoro particolarmente elevato.
4. Il sistema non sia critico.
5. Si possa associare al pattern Aggregate.

   Infatti gestire un dominio ad eventi senza la possibilità di avere un *aggregate root* è controproducente, in quanto non si avrebbe un "singolo punto" di generazione e applicazione di eventi.

   A mio avviso può essere consigliato implementare E.S qualora, oltre ad Aggregate, sia possibile implementare anche CQRS, semplificando di molto la gestione degli eventi.

   La sinergia tra questi due pattern (CQRS ed ES) infatti è molto stretta, la creazione di eventi e la lettura di questi (che si traduce nella ricostruzione di un aggregato) rispecchia perfettamente i ruoli di un command Endpoint e di un View Endpoint.

Segue una sezione di riferimenti bibliografici, intesi come una lista di documentazioni e indicazioni seguite.

Riferimenti bibliografici

[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 (visitato il 20/12/2021).

[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 (visitato il 11/01/2022).

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

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

[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 (visitato il 20/12/2021).

[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. Dic. 2018. URL: https://mincong.io/2018/12/03/exception-handling-in-jax-rs/ (visitato il 11/01/2022).

[15] Kiran Kumar. *Microservices with CQRS and event sourcing - dzone microservices*. Mag. 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.

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

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

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

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

[21] John Au-Yeung e 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/ (visitato il 11/01/2022).

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