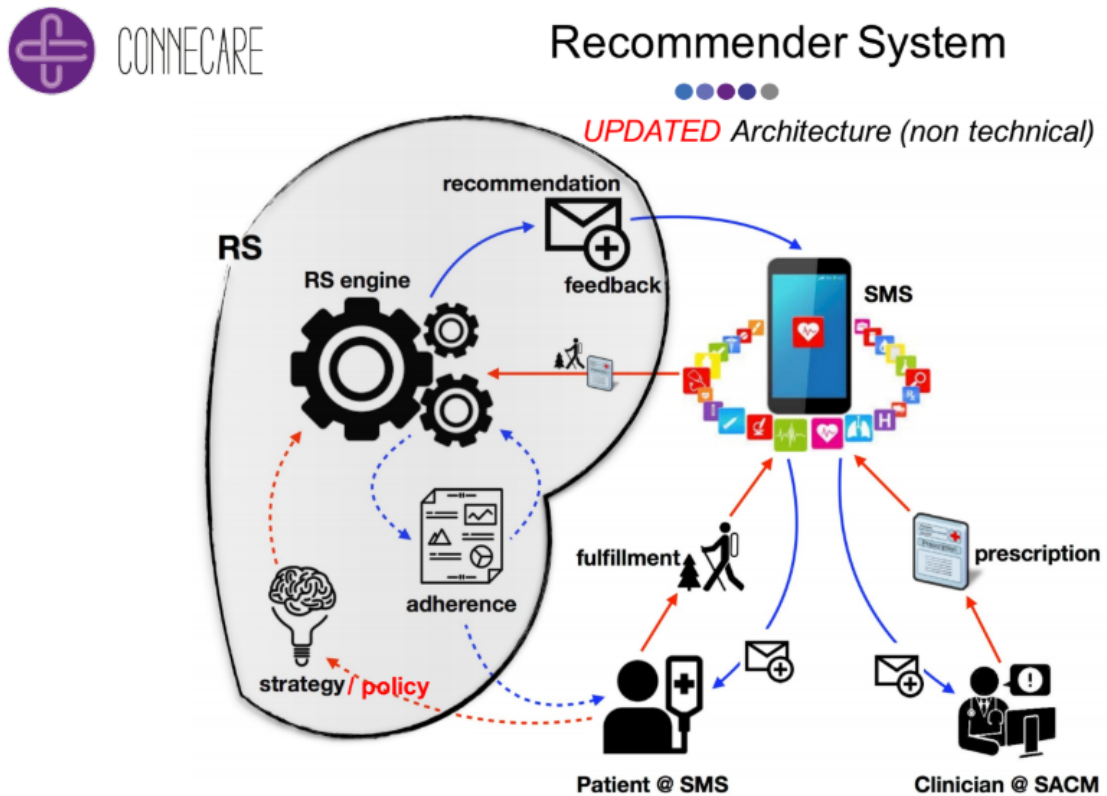

Microservizi con Vert.x

Creazione di un Recommender System



Professore: Andrea Omicini

Tutor: Stefano Mariani

Studente: Giacomo Zanotti

Matricola: 0000793982

Indice

1	Introduzione	2
1.1	Obiettivo del progetto	2
2	Analisi dei Requisiti	3
3	Background Tecnologico	3
3.1	Architettura a Microservizi: un (not too much) " <i>Deep Dive</i> "	3
3.2	Architettura REST e microservizi RESTful	5
3.3	Vert.x: not just another actors-like framework	7
4	Design delle API: Swagger	7
5	Recommender System a Microservizi	9
5.1	Architettura di Sistema	9
5.2	Architettura Software di un Microservizio: il Prescription Microservice	11
5.3	Esempio di un caso d'uso: PUT di una Prescription	14
5.3.1	Analisi	15
6	Conclusioni	20
6.1	Sviluppi Futuri	20
6.2	Problemi riscontrati	20
7	Guida tecnica	23
7.1	Import del progetto da BitBucket	23
7.2	Introduzione pratica su Vert.x	24
7.3	Service discovery	28
7.4	Aggiungere un nuovo Microservizio al Recommender System	30
7.5	Swagger	31
	Riferimenti bibliografici	37

1 Introduzione

Il recommender system è un prodotto healthcare che mette in stretta relazione il paziente con il proprio medico, cambiando il modo classico di portare avanti le procedure cliniche. Il sistema non si ferma alle semplici prescrizioni del clinico, ma sfrutta wearable technologies utilizzate dal soggetto in cura per un monitoraggio sugli obiettivi che deve conseguire. Il monitoraggio e le prescrizioni sono alla base dell'intero prodotto, introdotti nel software rispettivamente come accomplishment (o fulfillment) e prescriptions.

- **Prescription.** La ricetta del medico, in cui troviamo il codice identificativo del clinico assieme ad una lista precisa di goals da completare entro una certa data da parte del soggetto, anch'esso identificato da un codice univoco
- **Accomplishment.** I dati in arrivo dal paziente attraverso l'uso di wearable technologies (come un fitbit)

La misura delle prestazioni si ottiene con il calcolo dell'aderenza, in cui si comparano i dati numerici provenienti dal accomplishmentfulfillment con i traguardi da raggiungere della prescrizione. L'aderenza viene calcolata su due livelli, uno relativo alla singola prescrizione e l'altro su una history basata su un intero periodo di monitoraggio. A seconda del tipo di strategia che si può adottare per il calcolo dell'aderenza, si possono generare le Recommendations e i Feedback, rispettivamente per il paziente e per il medico. Le recommendations sono veri e propri consigli forniti al soggetto che lo avvertono dei suoi progressi verso i goals forniti dalle prescrizioni. In base al tipo di policy che viene adottata, verranno forniti dei messaggi di tipo WARNING, ENGAGEMENT, REWARD. I feedback vengono invece generati per il medico, che li può utilizzare per osservare come migliora o peggiora il proprio paziente, sia per quanto riguarda la prescrizione corrente (adherence level: il livello dell'aderenza rispetto agli obiettivi) e rispetto ad un profilo storico.

1.1 Obiettivo del progetto

L'obiettivo del progetto è la creazione di un sistema distribuito basato su un'architettura a microservizi per esporre le API del Recommender System. L'implementazione vera e propria di questo backend segue una fase di design delle API RESTful,

in cui vengono definiti tutti i dettagli: dai modelli degli oggetti da restituire come *HTTP response* fino ai parametri delle *HTTP requests*.

2 Analisi dei Requisiti

- **Microservizi con *Vert.x*** L'implementazione delle API del Recommender System deve seguire la direzione imposta dall'architettura a microservizi, utilizzando *Vert.x* come framework di supporto.
- **REST *best practices* per le API** Le API devono essere create utilizzando l'architettura REST, avendo particolare attenzione alle *best practices* consigliate e adottate dalla community.
- **Integrazione codice esistente** Il Recommender System è un prodotto già ben avviato, in cui sono state realizzate alcune parti, funzionanti e testate, e l'aggiunta delle REST API deve essere integrata con il codice esistente. I punti in cui è avvenuta l'integrazione riguarda la **modellazione dei dati** e le **strategie di business logic**. Infatti i dati che vengono maneggiati devono utilizzare classi e interfacce già implementate, mentre per la generazione di messaggi "ad hoc" di *recommendation* e *feedback* bisogna includere la business logic presente.

3 Background Tecnologico

Lo studio dei requisiti è stata una parte importante nella prima fase di progetto. In questa fase di indagine sulle scelte da adottare per raggiungere gli obiettivi è stata studiata l'Architettura a Microservizi[1], utilizzando articoli e documentazioni di progetti.

3.1 Architettura a Microservizi: un (not too much) "*Deep Dive*"

Architettura Le applicazioni classiche vengono progettate utilizzando procedimenti che separano la logica in vari moduli, ma impacchettati tutti assieme per

dare vita ad unico prodotto monolitico. Il risultato è un prodotto che con il tempo diventa troppo grande, e anche troppo delicato: aggiungere e togliere codice o tecnologie ha impatti sull'intero sistema. La possibilità di accedere al web da tantissimi tipi di dispositivi ha reso ancora più complicato la gestione di un traffico internet sempre più intenso: per questo motivo diventa necessario adottare design responsivi, reattivi e fault tolerant. L'Architettura a Microservizi è basata sulla suddivisione delle funzionalità del sistema in piccole unità, indipendenti, affidabili e loosely coupled. I vantaggi si possono osservare da vari punti di vista:

- **Necessità di progetto.** I servizi loosely coupled permettono un miglioramento netto dell'organizzazione e della gestione del lavoro, in cui piccoli team possono realizzare il proprio servizio in modo indipendente, senza creare conflitti.
- **Continuous Delivery.** Il prodotto non è statico, ma in continua evoluzione. Aggiungere e togliere funzionalità diventa molto più semplice.
- **Evolution of technology stack.** Come i gusti e le necessità dei consumatori cambiano, cambiano anche le tecnologie. I servizi loosely coupled permettono di aggiornarle o cambiarle con impatti nulli o minimi sul sistema.

Il Microservizio Un microservizio è un'entità che gestisce una (relativamente) piccola logica, indipendente e completa. La sua funzionalità è parte di un sistema più vasto, e nel soddisfare certe richieste dal client diventa necessario che più microservizi lavorino insieme. Un esempio può essere visto nel progetto: una richiesta HTTP per sapere qual è il livello di aderenza di un paziente rispetto alla prescrizione corrente, utilizza sia il microservizio dell'aderenza stessa che quello della prescrizione. I risultati di entrambi vengono poi combinati assieme per fornirne uno unico in risposta. Alla luce di ciò è indispensabile chiarire l'**Inter Process Communication**, ovvero le modalità con cui i microservizi comunicano tra di loro:

- **Synchronous communication: API consumption.** Un microservizio consuma le API di un altro per portare a termine una richiesta http.

- **Asynchronous communication: message passing.** Il *message passing* è un sistema noto di programmazione asincrona in cui vari thread comunicano tra di loro attraverso uno scambio di messaggi.

. Un sistema a microservizi generalmente non adotta un solo tipo di *IPC*, ma li combina assieme: una richiesta http è di natura sincrona, ma per velocizzare i tempi di risposta si possono adottare meccanismi basati su message passing.

API Gateway L'API Gateway è il punto di ingresso del sistema: gestisce ogni richiesta del client per ridistribuirla tra i vari microservizi, ottiene i risultati e li invia in risposta come unico. È una soluzione efficace per ogni sistema distribuito, perchè le richieste possono arrivare da ogni tipo di dispositivo con esigenze diverse: un client mobile e un client PC differiscono molto per capacità computazionale, dimensione dello schermo e user experience. Avere un unico entry-point significa avere la possibilità di creare API ad hoc per ogni tipo di client, semplificando molto la parte front end dell'applicazione e migliorando la user experience. Sfruttando il fatto che il Gateway è l'unico punto di ingresso del sistema, è possibile anche applicare politiche di load-balancing per gestire il traffico delle chiamate, servire contenuti statici e impostare delle strategie per il caching dei dati. Un Api Gateway implementato male può portare a due tipi di **bottleneck**: il primo riguarda il volume delle chiamate in ingresso che può diventare ingestibile e il secondo riguarda il bottleneck implementativo, cioè ogni volta che viene aggiunto o cambia un microservizio bisogna aggiornare il Gateway.

3.2 Architettura REST e microservizi RESTful

REST REST significa REpresentational State Transfer e propone metodi per creare servizi web indipendenti dalla tecnologia o dalla piattaforma e si basa su sei principi:

1. **Uniform Interface.** È necessario adottare un'interfaccia uniforme per la manipolazione di tutte le risorse, identificabili attraverso un URI logico e unico, con sufficienti informazioni. Una risorsa è il risultato di un'astrazione dei dati da manipolare, ovvero una rappresentazione. In particolare, l'accesso, la modifica o la creazione delle stesse devono avvenire attraverso l'utilizzo di protocolli conosciuti e adottati dai programmatori, come i metodi HTTP.

2. **Client-Server.** Client e server devono essere in grado di evolversi separatamente, senza che l'uno dipenda dall'altro. Il client ad esempio ha solo bisogno di sapere la locazione delle risorse, mentre il server deve solo fornire dati al momento della richiesta.
3. **Stateless.** Il server non contiene dati sullo stato del client, ogni richiesta viene trattata come nuova.
4. **Layered System.** utilizzando un'interfaccia comune tra client e server, è possibile eseguire tutti vari tipi di operazioni su server diversi.
5. **Cacheable.** La performance del sistema aumenta enormemente se si utilizza la cache del browser. Certe risorse vengono riusate spesso, e sfruttare la cache significa annullare qualunque latenza tra client e server.
6. **Code On Demand (Optional).** È possibile estendere le funzionalità del client attraverso il server. Il client può eseguire il download e successivamente eseguire il codice che arriva dal server.

RESTful Microservices La natura multi-platform dei servizi REST è ideale in un contesto di microservizi. Ogni microservizio contiene le proprie API realizzate secondo le indicazioni REST, utilizzando quindi le regole e le *best practices*[3] adottate dalla comunità.

3.3 Vert.x: not just another actors-like framework

Vert.x[4] è un framework vasto e poliglotta, e copre tantissimi tipi di ambiti. La sua unità centrale è **Vert.x.core**, su cui si appoggiano numerose estensioni (come **Vert.x.web**, per la creazione delle API e dei server).

Vert.x.core possiede molte api per permettere la realizzazione di applicazioni event driven, basate su Verticles e scambio di messaggi tramite l'event-bus.

Verticle I verticles costituiscono le fondamenta di ogni applicazione scritta con Vert.x. Grazie ai verticles è possibile realizzare progetti in modo single-threaded, lasciando che sia il framework a fare tutto il lavoro di multi-threading e coordinamento. Possono essere pensati come degli attori di un actor-model, ma i messaggi in questo caso vengono scambiati sfruttando l'event-bus.

EventBus L'event-bus è il sistema nervoso di Vert.x, e funziona attraverso l'utilizzo di indirizzi, ad ognuno dei quali è associato un handler da implementare. Inviare messaggi ad un indirizzo fa sì che l'handler si attivi. L'invio di messaggi può essere sia point-to-point (il messaggio interessa un solo handler associato all'indirizzo) oppure publish-subscribe (il messaggio viene replicato per ogni handler associato all'indirizzo).

4 Design delle API: Swagger

Per un design delle api efficace è stato utilizzato **Swagger**¹[2], uno strumento che permette di modellare il dominio del problema utilizzando JSONSchema. JSONSchema è un linguaggio in grado di validare le strutture dati JSON, e assieme

¹Swagger è un tool online, che favorisce la collaborazione fra programmatori per il design dei API secondo le best practices REST

all'editor online swagger impone l'utilizzo delle specifiche **OPENAPI 3.0**, un format standard per la creazione di API REST. Le API sono state realizzate seguendo

```
/rs/prescriptions/{prescrUUID}:
put:
  tags:
    - Prescription
  description: PUT method to update a prescription
  parameters:
    - in: path
      name: prescrUUID
      schema:
        type: string
        format: uuid
        example: d290f1ee-6c54-4b01-90e6-d701748f0851
        required: true
      description: uuid of the prescription
  requestBody:
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Prescription'
  responses:
    '200':
      $ref: '#/components/responses/SuccessfulRequest'
    '400':
      description: prescription was not found
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/PrescriptionError'
    '403':
      $ref: '#/components/responses/Unauthorized'
    '404':
      $ref: '#/components/responses/NotFound'
```

Figura 1: Esempio del design di una chiamata http *PUT* per modificare una prescription

il design imposto da **Swagger**, che definisce le seguenti caratteristiche:

- **URL della risorsa.** L'indirizzo della risorsa deve essere ben definito e si consiglia di usare le REST best-practices per utilizzare un format ben comprensibile.
- **Parametri della chiamata.** I parametri delle risorse devono essere descritti dettagliatamente, dando specifiche sul nome, il tipo di valore e se sono *path*

variables oppure *request variables*.

- **Definizione della response.** È indispensabile fornire dettagli sul tipo di valore di ritorno utilizzando gli header con specificato il *media/type*.
- **Gestione degli errori.** Gli errori possono accadere e ce ne sono di vari tipi, e dipendono sia da come è stata inviata la richiesta (errori nei parametri, o semplicemente "spelling" dell'URL) o se la risorsa non è presente. Con Swagger è possibile aggiungere i vari tipi di errori fornendo sia la descrizione che il messaggio da restituire.
- **Definizione dei modelli.** Allo stesso modo è possibile definire il modello dei dati da restituire al client, con descrizioni dettagliate su tipi e variabili

5 Recommender System a Microservizi

5.1 Architettura di Sistema

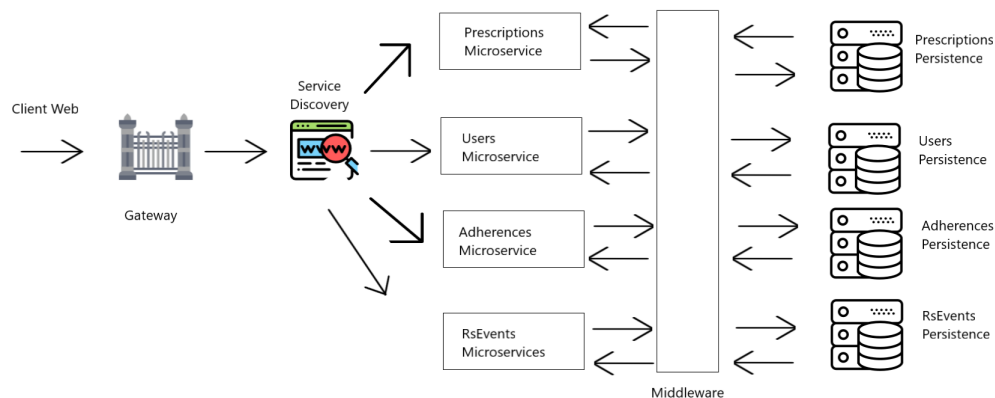


Figura 2: Architettura di sistema

Divide et Impera Il funzionamento complessivo del Recommender System è stata suddiviso in varie unità logiche, ognuna gestita da un microservizio. Questi componenti espongono le proprie api al sistema, hanno una propria logica di

persistenza e dialogano tra di loro per ottenere dati già pronti per l'utilizzo. L'elaborazione della risposta da fornire al client funziona attraverso una combinazione di due tipi di *inter process communication*: consumazione delle API di un microservizio (synchronous request-response) e scambio di messaggi. Ogni singolo servizio raccoglie i risultati in modo asincrono da altri microservizi, e una volta pronti vengono inviati come response.

- **Prescriptions.** Il microservizio tiene traccia di tutte le prescrizioni e fornisce varie funzioni di utilità che vengono utilizzate dagli altri microservizi. È fondamentale per il calcolo dell'aderenza e viene utilizzato anche dai servizi di Recommendation e Feedback.
- **Users.** L'endpoint del sistema per quanto riguarda la gestione dell'utente. Proprio a questo servizio arrivano i dati dei vari accomplishment, e fornisce anche metodi per ottenere dati più complessi come i dettagli di un paziente.
- **Adherences.** Un microservizio che esegue operazioni logiche per il calcolo dell'aderenza di un paziente rispetto alla sua prescrizione. Ottenendo i dati degli obiettivi e i dati degli accomplishment, utilizza una strategia di computazione per restituire il valore in percentuale dell'attività svolta dall'utente
- **RSEvents.** Sono due microservizi che svolgono attività molto simili ma per richieste diverse. Il loro funzionamento richiede dati da tutti i microservizi per un corretto funzionamento della *policy* con cui si creano i messaggi per utenti (Recommendation) e medici (Feedback).

API Gateway e relazione con i microservizi Il Gateway è l'unico entry point del sistema, quindi è in grado di raccogliere tutte le richieste dal client web, per poi ridistribuirle tra i vari microservizi. Questo permette di semplificare molto il funzionamento a lato client: una richiesta per una recommendation è in realtà il lavoro di più microservizi, ma viene fatta solo una singola chiamata al Gateway, che colleziona i risultati una volta pronti e li restituisce tutti assieme. L'API Gateway utilizza Vert.x per trovare tutti i microservizi del sistema grazie ad un meccanismo di *Service Discovery*. Il client esegue la richiesta sempre su un unico

server, mentre il Gateway sfrutta il service discovery per ottenere l'indirizzo del server e della porta dei microservizi.

Service Discovery Il *Service Discovery* è un servizio che permette di trovare tutti i microservizi del sistema in modo dinamico a run-time. Aggiungere funzionalità è molto semplice, ma se il sistema diventa troppo grande diventa impossibile tenere traccia di tutti i servizi. Ognuno di questi è un server web con un proprio indirizzo e porta su cui ascolta le richieste http, ed è quindi fondamentale fare il deploy avendo cura di non creare conflitti (es. due microservizi, essendo due server web, non possono ascoltare sulla stessa porta quando hanno lo stesso indirizzo IP). Ogni microservizio al momento della creazione di una sua istanza "pubblica" sul service discovery le proprie specifiche: il nome, l'indirizzo e la porta. Una volta pubblicato, è esposto al sistema attraverso lo stesso service discovery; quest'ultimo può essere interrogato per cercare il microservizio fornendo il suo nome, ottenendo indirizzo e porta per creare una richiesta http.

Message-passing based Middleware Il middleware è un componente dell'architettura che gestisce le modalità di creazione o lettura dei dati e la comunicazione tra i vari microservizi. Il funzionamento del middleware è basato sullo scambio di messaggi. Ogni microservizio in qualunque momento può spedire una richiesta per accedere alla propria persistenza o a quella di un altro, inviando un messaggio specificando un indirizzo.

5.2 Architettura Software di un Microservizio: il Prescription Microservice

Le componenti software del microservizio Il prescription microservice gestisce tutte le funzionalità del sistema per quanto riguarda le prescrizioni del medico

- ***PrescriptionRESTAPIService***. È un Verticle che contiene l'implementazione della parte Server del microservizio.
- ***PersistencePrescriptionsService***. È il Verticle che gestisce la comunicazione tra i microservizi e la persistenza.

- **Persistenza.** La persistenza è definita in modo loosely coupled dal microservizio stesso tramite le sue interfacce.

Server e comunicazione con il Gateway La parte Server del microservizio è costituita da un verticle (*PrescriptionRESTAPIService*), che crea al suo interno un server web ascoltando su una porta non utilizzata da altri microservizi.

Il server rimane in ascolto delle richieste http provenienti dal client e viene coinvolto solo per le chiamate relative solamente alle *prescription*: rimanendo in linea con il principio alla base dell'architettura a microservizi, non è possibile per un server gestire delle risorse che vanno al di fuori della proprio area di competenza.

In questo caso l'inter-process communication policy adottata è pura comunicazione sincrona mediante richieste http, e gli end-point interessati non sono altro che il microservizio stesso e il suo client, il Gateway: il Gateway manda la richiesta, e rimane in attesa finchè non arrivano tutti i dati richiesti.

Le API del Prescription Microservice sono gestite da un *Router*², che gestisce varie *routes* mappando gli URL delle risorse a degli *handler*. Questi handler contengono la completa implementazione delle API, e si attivano solo quando arriva la chiamata http corretta.

Il verticle quando viene istanziato oltre a preparare il router e registrare tutte le routes comunica al sistema mediante il *Service Discovery* la sua avvenuta creazione, specificando il nome del proprio servizio, la porta e l'indirizzo. In questo modo potrà essere recuperato dal Gateway oppure recuperato anche in altre parti del sistema. Dato che questa azione viene eseguita da ogni Verticle che implementa la logica server, è stata creata una classe astratta: *BaseMicroService*. Il *PrescriptionRESTAPIService* eredita vari metodi già implementati, tra cui quelli per creare un server e pubblicazione a run-time sul *Service Discovery*.

È compito del server anche creare il Verticle per la gestione dei dati memorizzati dalla persistenza.

Microservizio come attore Il *PrescriptionRESTAPIService* è dissociato dalla gestione dei dati, e sfrutta un verticle di supporto per comunicare con gli altri

²Un router è un oggetto fornito da Vert.x. Per approfondimenti, vedere il caso d'uso o l'appendice

microservizi e per gestire la persistenza, il *PersistencePrescriptionsService*. L'utilizzo di un nuovo verticle ha come impatto immediato la caratterizzazione loosely coupled del servizio, che permette di aggiungere o modificare le funzionalità del microservizio stesso.

Un altro vantaggio non indifferente è la facilità di lettura e una migliore comprensione del codice, per intervenire con sicurezza nella modifica.

Attraverso il verticle, il microservizio può essere pensato proprio come un vero attore di un *actor system*, e comunica con gli altri microservizi mediante lo scambio di messaggi. Sfruttando la natura reattiva di questo modello, è possibile creare un sistema efficiente e anche resiliente, evitando le complessità derivanti dall'uso di chiamate sincrone (come ad esempio l'interruzione della connessione). Lo scambio di messaggi avviene attraverso il middleware, l'*Event-bus*³ di Vert.x.

Alla stessa maniera del *PrescriptionRESTAPIService*, il *PersistencePrescriptionsService* ha accesso all'Event-Bus su cui registra a degli indirizzi gli handler delle operazioni da eseguire sulla persistenza.

Il *PrescriptionRESTAPIService* utilizza questi indirizzi per fare richiesta di una risorsa (può essere la propria o quella di un altro microservizio), inviando un comando *send* sull'*Event-bus* con un messaggio come payload, contenente le informazioni per eseguire, ad esempio, il recupero di una persistenza salvata in un certo periodo di tempo. Il *PersistencePrescriptionsService*, avendo registrato gli handler agli indirizzi in uso, esegue le operazioni al momento dell'arrivo del messaggio: viene attivato l'handler corrispondente e usa il contenuto per ottenere le informazioni corrette per recuperare le informazioni dalla persistenza. Una volta eseguita l'operazione, il risultato viene restituito dal sempre attraverso con lo scambio di messaggio attraverso un comando *reply*.

Persistenza La persistenza è unica e ci si può accedere solo mediante il *PersistencePrescriptionsService*. È una scelta di design che segue il principio dell'architettura adottata: se la persistenza è unica per microservizio, aggiungere nuove funzionalità, cambiare il modello di persistenza dei dati o modificare i parametri di ricerca diventa un'operazione ad impatto zero per il sistema.

³Guardare sezione precedente per informazioni sull'Event-bus

La persistenza è unica, ma comunque esposta agli altri microservizi sfruttando lo scambio di messaggi: un microservizio può ottenere i dati da un'altra persistenza semplicemente utilizzando la modalità vista in precedenza con il message-passing. La modalità di persistenza adottata per il caso d'uso è solo una memorizzazione temporanea in memoria ⁴, ma comunque espandibile grazie all'interfaccia che può essere implementata diversamente a seconda dei casi. La *IPrescriptionPersistence* contiene vari metodi per effettuare operazioni base sui dati, ed utilizza un pattern *Strategy* ⁵ per gestire le varie modalità di recupero di una o più prescrizioni, definendo anche un limite temporale.

5.3 Esempio di un caso d'uso: PUT di una Prescription

All'inizio c'è il deploy Nella fase iniziale, avviene il *deploy* del sistema, in cui vengono istanziati i verticles che compongono i microservizi. Le azioni in questa fase sono la creazione del server, la pubblicazione sul service discovery e la creazione del verticle di supporto per la comunicazione tra i vari microservizi e per la persistenza.

La best-practice in questo caso è creare con ordine prima il server: se il server non è attivo non è possibile accederci tramite il servizio di discovery o solo per ottenere i dati memorizzati.

Nonostante la creazione di un Verticle sia prossocchè istantanea ed indolore, non lo è per quella di un server o per la pubblicazione sul service discovery, che può richiedere tempi diversi o addirittura generare errori.

In questo caso per assicurarsi che venga prima creato il server, si usa il *chaining* ⁶ di *Futures*, oggetti che contengono i risultati delle computazioni asincrone ancora incomplete. L'ordine delle istruzioni è rispettato grazie all'operazione *compose*, che permette di eseguire un comando asincrono solamente se il precedente ha avuto successo. Tutto questo deve essere eseguito nel metodo *start* di un Verticle, ⁷. Una

⁴Per mancanza di tempo, non è stato possibile implementarne una vera adottando un vero e proprio database

⁵Il pattern strategy è utilizzato per cambiare il comportamento di una classe a run-time

⁶Il chaining è una tecnica per eseguire con sequenzialità operazioni asincrone

⁷Nella sezione "Guida alla creazione di un Microservizio" ci sono tutti i dettagli

```

@Override
public void start() {
    createAPIServer(router)
        .compose(server -> publishHTTPEndPoint(
            SERVICE_NAME, HOST, PORT, "/" + API_SERVICE_TYPE))
        .setHandler(event ->
            vertx.deployVerticle(new
                PersistencePrescriptionsService(
                    new TempPrescriptionPersistenceImpl())));
}

```

Figura 3: Esempio di una composizione di futures dei metodi asincroni *createAPIServer* e *publishHTTPEndPoint*

volta che i server sono stati creati, correttamente pubblicati sul service discovery indirizzo e porta e creato il verticle per l'inter-process communication, è possibile effettuare le chiamate http.

5.3.1 Analisi

Il client web deve mandare la richiesta direttamente al server Gateway e non al microservizio stesso, rendendo più semplice l'utilizzo delle API.

`http://localhost:8080/rs/prescriptions/{uuIdPrescription}`

Funzionamento del Gateway Il Gateway può essere immaginato come un server *proxy* (`http://localhost:8080` è il suo indirizzo IP con porta), un intermediario tra il client web e il server del microservizio.

1. **Ricerca del servizio.** Viene eseguito il parsing dell' URI per trovare il microservizio. Se si nota l'url precedente, si può vedere che all'inizio c'è sempre la sigla **rs** (**Recommender System**) ed è alla radice di ogni chiamata http. A seguire invece c'è sempre il nome del microservizio a cui si vuole accedere, e il service discovery lo usa per ottenere il suo indirizzo ip e porta.

2. **Creazione di un client.** Il Gateway crea un nuovo client che manda una nuova richiesta http, ma utilizzando indirizzo IP e porta del microservizio. Nel caso in analisi il nuovo url che viene usato per la chiamata http è:

`http://localhost:8081/prescriptions/{uuIdPrescription}`⁸

3. **Attesa della response.** La response del microservizio ritorna al client creato dal Gateway e solo allora viene passata al client web da dove è iniziata la chiamata http.

⁸L'indirizzo IP è sempre localhost perchè il deploy è solo locale, ma potrebbe essere un qualunque indirizzo IP, in un qualunque server.

Arrivo della chiamata al Server Il Server utilizza un *Router* di Vert.x per memorizzare tutti i vari URL delle risorse. Ad ognuno di loro è associato un handler, dove viene eseguito il codice per creare una response. All'interno della

```
private void apiPutPrescription(RoutingContext routingContext) {
    //first part of the handler
    UUID uuidPrescr = null;
    try {
        uuidPrescr =
            UUID.fromString(routingContext.pathParam(PATH_PARAM_UUID));
    } catch (IllegalArgumentException e) {
        routingContext.response().setStatusCode(400)
            .putHeader("content-type", "text/plain")
            .end("UUID not valid");
    }
}
}
```

Figura 4: Handler della chiamata http

response si possiede un riferimento alla richiesta http tramite la *Routing Context*, un oggetto fornito da Vert.x. Si può utilizzare per ottenere gli header, i parametri di richiesta o anche il *body*, come nel caso di una chiamata PUT o POST. Lo stesso oggetto è usato per creare una response, come si può notare dalla notazione: `routingContext.response()`. Viene innanzitutto effettuata un'analisi sulla correttezza delle variabili di richiesta (in questo caso, l'uuid della prescription), utilizzando la gestione delle eccezioni di Java con il costrutto `try-catch`. Se l'uuid non è corretto, viene subito inviata una response di errore al Gateway che la fornirà nuovamente al client web.

In mancanza di errori, viene inviato un messaggio sull'EventBus attraverso il comando *send*, visibile nello snippet del codice.

Il messaggio inviato non è altro che un oggetto che rappresenta una prescription, ma in formato JSON⁹.

```
private void apiPutPrescription(RoutingContext routingContext) {
    //second part of the handler
    if (uuidPrescr != null) {
        vertx.eventBus().send(EventBusAddresses.PUT_ADDRESS_PRESCRIPTIONS,
            prescrJson,
            new DeliveryOptions()
                .addHeader("prescriptionUUID", uuidPrescr.toString()),
            reply -> {
                if (reply.succeeded() && reply.result().headers()
                    .get("status").equals("success")) {
                    routingContext.response().setStatusCode(201)
                        .putHeader("content-type", "application/json")
                        .end("resource successfully created");
                } else if (reply.succeeded()) {
                    routingContext.response()
                        .setStatusCode(400).putHeader("content-type",
                            "text/plain")
                        .end(reply.result().body().toString());
                }
            });
    }
}
```

Figura 5: Handler della chiamata http

Il Verticle PersistencePrescriptionService Un messaggio per essere inviato deve specificare nel metodo *send* un indirizzo, in questo caso una semplice stringa. Oltre all'indirizzo, viene definito un handler per l'eventuale risposta, ma

⁹ Il modello JSON dei dati da salvare o restituire in response è stato definito con *Swagger*

utilizzarlo è facoltativo. Nel caso in analisi viene utilizzato perchè il *PersistencePrescriptionService* deve rimandare indietro un messaggio di conferma della correttezza dell'operazione o uno con le possibili eccezioni, generate dal parsing dei modelli JSON per ottenere gli oggetti Java corrispondenti.

```
private void handlePutPrescription(Message message) {
    try {
        persistence.saveFeed(ParseUtils
            .parsePrescription((JsonObject)message.body()));
        DeliveryOptions options = new DeliveryOptions();
        options.addHeader("status", "success");
        message.reply("prescription is updated", options);
    } catch (Exception e) {
        e.printStackTrace();
        message.reply(e.getMessage(), new
            DeliveryOptions().addHeader("status", "fail"));
    }
}
```

Figura 6: Esempio di salvataggio della modifica di una prescription, utilizzando l'interfaccia della persistenza

Il verticle utilizza l'Event-bus per creare un *consumer*: un consumer non è altro che un handler che viene associato all'indirizzo. Il messaggio viene preso dall'handler ed estratto il suo contenuto. Utilizzando l'interfaccia della persistenza, viene salvato la prescription in memoria, e le possibili eccezioni vengono raccolte e rimandate indietro al mittente tramite messaggio, dove viene riportato l'errore in formato stringa.

Se invece le eccezioni non ci sono, allora il salvataggio dei dati è avvenuto correttamente e si invia un messaggio di successo.

Ritorno al Gateway Quando il *PersistencePrescriptionService* completa le operazioni, rimanda indietro al mittente il risultato che viene raccolto dal server e successivamente inviato al Gateway tramite *response*. Se il messaggio con il risul-

tato è di successo, lo status code è 200, altrimenti c'è un errore, e la response ha lo status code 400.

6 Conclusioni

6.1 Sviluppi Futuri

La natura loosely coupled dell'architettura adottata permette di modificare o aggiungere funzionalità agevolmente, con l'unico requisito di aggiornare il Service Discovery. Al di là dell'aggiunta di nuove funzionalità, possono essere effettuati miglioramenti su vari livelli:

- **Comunicazione tra i microservizi.** La comunicazione è necessaria per richiedere i dati necessari per esempio al calcolo di una recommendation. Anche se la modalità implementata con puro scambio di messaggi funziona correttamente, può risultare complessa e a volte più difficile da modificare. In questo caso si può usare il *Service Proxy* di Vert.x, che permette di esporre un servizio (come appunto, la persistenza) sull'event bus, senza scrivere codice "boilerplate" (tener traccia degli indirizzi su cui mandare i messaggi, fare il parsing di oggetti ecc.). La raccolta dei dati diventa semplicemente un chiamata di un metodo, ci penserà poi Vert.x a gestire tutto.
- **Persistenza.** La persistenza di un microservizio è esposta attraverso un interfaccia e implementata da classi *temp*, che memorizzano i dati in liste interne. Sarebbe più interessante aggiungere un vero e proprio database per microservizio, magari anche diverso (un documentale come *MongoDB* o un relazionale basato su query *SQL*).

6.2 Problemi riscontrati

Studio e implementazione dell'architettura a Microservizi L'architettura a microservizi è un trend molto nuovo, e non esistono delle vere e proprie linee guida o delle *best practices* come per l'architettura REST.

È stato necessario un periodo di studio dello stato dell'arte, cercandone di capire i principi per poi passare alla creazione del codice.

La comprensione del modello non è complessa e gira attorno a concetti di certo non nuovi, come la suddivisione del problema in unità logiche. La vera sfida è stata la parte implementativa dell'architettura:

- **Creazione di un API Gateway.** da un punto di vista teorico, non è complicato, ma quando entra in gioco la chiamata dei microservizi si incomincia a notare le difficoltà. Essendo un progetto accademico, l'implementazione è semplificata, ma può essere ampliato di molto, creando un vero e proprio servizio di caching dei dati più frequenti o aggiungendo un possibile servizio di server side rendering.
- **Service discovery.** Creare il service discovery da zero è molto complesso e richiede tempo: deve tener traccia di tutti i microservizi in piedi e saper ottenere tutti i loro indirizzi IP con porta.
- **Inter-process communication.** Anche in questo caso è impossibile creare un servizio di inter process communication da zero, in particolare basato su message passing.

Questi problemi sono stati risolti grazie a Vert.x. Vert.x è stato costruito proprio per creare sistemi a microservizi: contiene vari elementi built-in già pronti per l'utilizzo, come l'inter-process communication con scambio di messaggi attraverso l'*Event bus*, il *Service Discovery*, il *Router* per le API, e la possibilità di creare un server web. Il framework ha permesso di accelerare la creazione del sistema, permettendo di focalizzarsi sui dettagli implementativi, sulle modalità di comunicazione tra i microservizi e sull'interazione tra gateway e microservizi

Programmazione asincrona Nonostante Vert.x sia semplice da usare, ci sono comunque problematiche della programmazione asincrona che bisogna affrontare. Molti dati vengono da più servizi diversi, con l'effetto di creare in certi punti le famose *pyramid of doom*[5], in cui sono innestati l'uno dentro l'altro gli handler con il risultato della computazione. Al tal fine, dove è stato possibile, è stato usato un meccanismo a base di Future per chiamare contemporaneamente più servizi di persistenza, collezionare i dati e rimandarli al Gateway. Lo scambi di messaggi come modalità di programmazione asincrona permette di implementare con molta efficienza tutto lo scambio di dati, ma ne aumenta la difficoltà.

Swagger Swagger è stato uno strumento utile per approfondire i requisiti funzionali delle API, in modo cooperativo con il committente. Impostando il design delle API precedentemente alla parte pratica del progetto ha aiutato significativamente ad accelerare il lavoro, evitando incomprensioni fin dall'inizio. Nonostante questo aspetto positivo, l'uso di Swagger ha comportato vari problemi:

- **Comprensione di JSONSchema.** JSONSchema è un linguaggio molto simile al JSON, e possiede varie regole di sintassi come un linguaggio di programmazione vero e proprio: capire come si descrivono i parametri, vedere come aggiungere i riferimenti non è stato semplice.
- **Manutenzione delle API.** È normale che durante il procedimento qualcosa cambi, che sia il path o anche cose più delicate come i parametri, i tipi di errori, i response body ecc. Se c'è elevata discrepanza tra swagger e codice lo strumento diventa pressochè inutile, perchè gli altri programmatori devono guardare il codice.

Swagger è utile in un team, con una buona divisione di lavoro. Se il progetto è in singolo, lo strumento diventa troppo pesante e tedioso da mantenere aggiornato.

7 Guida tecnica

7.1 Import del progetto da BitBucket

Il progetto è stato importato da BitBucket su IntelliJ IDEA, un IDE per lo sviluppo di applicazioni in Java.

1. **Clone del progetto.** Utilizzando un qualunque client git¹⁰, eseguire il comando `git clone` con il link alla repo: `https://GiacomoZanotti@bitbucket.org/agentgroup/rs-physical-activity.git`. Il risultato è una cartella con nome **rs-physical-activity**. La cartella su cui si lavora è una sub-directory al suo interno con nome **RS-physical-activity**.
2. **Import del progetto.** Importare il progetto come progetto gradle, selezionando la cartella del punto 1.
3. **Checkout sul branch.** Utilizzare il comando git dal client (oppure utilizzare intellij) per andare sul branch del progetto.

```
git checkout students/zanotti
```

IntelliJ installerà tutte le dipendenze da Gradle, comprese quelle di Vert.x.

Se si vuole creare un nuovo progetto, andare su *File* → *New* → *Project*, impostare il project sdk selezionando la cartella dove si trova l'installazione di java sdk, e infine selezionare Gradle. Creato il progetto vuoto, aggiungere le dipendenze di Vertx

```
compile "io.vertx:vertx-core:$vertxVersion" //vert.x core contiene  
    gli elementi base di Vert.x  
compile "io.vertx:vertx-web:$vertxVersion" //vert.x web per  
    implementazione api e creazione server  
compile 'io.vertx:vertx-service-discovery:3.5.3'
```

¹⁰Io ho utilizzato git bash per windows, il download è disponibile cliccando qui

```
//questi ultimi tre servono nella code generation di classi proxy
    per evitare di usare direttamente lo scambio di messaggi,
    facoltativo
compile 'io.vertx:vertx-service-proxy:3.5.3'
compile "io.vertx:vertx-service-proxy:3.5.3:processor"
compileOnly 'io.vertx:vertx-codegen:3.5.3'
```

7.2 Introduzione pratica su Vert.x

Si parte sempre da un Verticle Il punto di partenza è sempre un Verticle. Quando si utilizza Vert.x il codice viene scritto come un insieme di più Verticle. Crearne uno è molto semplice, e basta estendere la classe *Abstract Verticle*. I me-

```
public class MyVerticle extends AbstractVerticle {

    // Called when verticle is deployed
    public void start() { }

    // Optional - called when verticle is undeployed
    public void stop() {}

}
```

Figura 7: Esempio di un Verticle con i metodi start e stop

todi *start()* e *stop()* sono metodi che vengono eseguiti rispettivamente al momento della creazione di un'istanza (fase di *deploy*) e al momento in cui viene distrutta (fase di *undeployment*). Ci sono vari tipi di Verticle che si possono realizzare, ma la cosa importante è che ognuno ha accesso all'oggetto *Vert.x*. Questo permette di sfruttare tutte le potenzialità del framework: avere accesso all'EventBus, creare i server e i client, utilizzare il service discovery e tanto altro. Per dar vita al Verticle, basta utilizzare il *deploy*:

```
public class Deployer extends AbstractVerticle{

    public static void main(String[] args){

        Vertx vertx = Vertx.vertx(); //riferimento all'oggetto di Vertx
        vertx.deployVerticle(new MyVerticle());

    }

}
```

Figura 8: Esempio di deploy

EventBus L'EventBus può essere pensato come un middleware, in grado di creare canali di comunicazioni tra vari punti del sistema, mediante lo scambio di messaggi.

I messaggi per essere inviati hanno bisogno di un **indirizzo**. Non è niente di particolare, è semplicemente una stringa con qualunque valore. Nel creare un'applicazione, è comunque consigliabile dare un significato al nome dell'indirizzo, per facilitare la comprensione ¹¹. Il messaggio viene ricevuto da un *handler*, che deve essere registrato allo stesso indirizzo su cui è stato inviato il messaggio.

Esempio di invio e ricezione di un messaggio Per inviare un messaggio ed eseguire del codice alla ricezione basta seguire tre semplici passaggi:

1. **Il riferimento dell'EventBus.** Per ottenere il riferimento all'EventBus si utilizza l'oggetto `vertx`, che viene fornito dal Verticle.

¹¹Il codice deve essere scritto in modo tale da "spiegarsi da solo", senza riempirlo di commenti

```
//MyVerticle.java
public class MyVerticle extends AbstractVerticle{

    @Override
    public void start(){
        EventBus eb = vertx.eventBus();
    }
}
```

Figura 9: Il codice di funzionamento va scritto nel metodo *start* del Verticle

2. **Invio del messaggio.** Basta utilizzare il riferimento del punto 1, creare un indirizzo, e passare il contenuto del messaggio:

```
//MyVerticle.java
public class MyVerticle extends AbstractVerticle{

    @Override
    public void start(){
        EventBus eb = vertx.eventBus();
        eb.send("this string is the address", new JsonObject().
            put("message", "Hi there"), reply->{
                //the reply to the message sent is handled here...
            });
    }
}
```

Figura 10: Il contenuto del messaggio è un oggetto `JsonObject`, incluso nella libreria di Vert.x

3. Registrazione dell'handler ed esecuzione del codice

Complicando leggermente l'esempio, in un nuovo verticle registriamo l'handler per l'indirizzo. Ogni volta che verrà inviato il messaggio all'indirizzo "this string is the address", verrà stampato sul terminale il messaggio.

```

//MySecondVerticle.java
public class MySecondVerticle extends AbstractVerticle{
    @Override
    public void start(){
        EventBus eb = vertx.eventBus();
        eb.consumer("this string is the address",handler->{
            Message message = handler.body();
            System.out.println("I have received message: "+message);
            //handler.reply("Hi, how are ya?")
        });
    }
}

```

Figura 11: Registrazione dell'handler mediante *consumer*

Il contenuto dei messaggi è un valore primitivo (*string*, *integer* ecc.), per creare messaggi con oggetti più complessi si può usare *JsonObject*.

Creazione di un Backend Per creare un server http si utilizza Vert.x Web, che contiene tante utilities per la programmazione web, tra cui il già citato *Router*. Si può già implementare in pochissimo tempo un server con delle API semplici.

```

public class Server extends AbstractVerticle{

    public static void main(String[] args){
        Vertx vertx = Vertx.vertx();
        vertx.deployVerticle(new Server());
    }
    @Override
    public void start(){
        Router router = Router.router(vertx);
        //Body handler per richieste PUT o POST ("/*" signigica "any
        path")
        router.route("/").handler(BodyHandler.create());
    }
}

```

```

router.post("/postpath").handler(routingContext->{
    routingContext.response().setStatusCode(201)
    .end("resource created!");
});

router.get("/path1/path2/:pathVariable").handler(routingContext->{
    routingContext.response().setStatusCode(200)
    .end("get request successfull");
});
//creazione del server
vertx.createHttpServer().
    requestHandler(router::accept).
    listen(8080);
}
}

```

In questo caso, una volta eseguito il run, se si digita sulla barra del browser

`http://localhost:8080/path1/path2/sono_una_path_variable`

comparirà il messaggio "get request successfull".

7.3 Service discovery

Il service discovery si può utilizzare sempre attraverso l'oggetto Vert.x all'interno di un verticle. Prima si pubblicano i dettagli del server (indirizzo e porta) con un nome.

```

//Once created, server's address and port can be published on
Service Discovery
vertx.createHttpServer().
    requestHandler(router::accept).
    listen(8080, result->{
        if(result.succeeded()){
            //Riferimento al ServiceDiscovery
            ServiceDiscovery discovery =
                ServiceDiscovery.create(vertx)

```

```

//Creazione di un record Http
Record record =
    HttpEndpoint.createRecord("server",
        "localhost", 8080, "/*");
//pubblicazione del record
discovery.publish(record,event->{
    if(result.succeeded()){
        System.out.println("Server saved on
            discovery!");
    }
});
}
});

```

In un altro punto del sistema, per fare una richiesta http all'indirizzo `http://localhost:8080/path1/path2` basta ottenere nuovamente un riferimento del Service Discovery e cercare il Record con il nome con cui è stato registrato:

```

discovery.getRecord(new JsonObject().put("name", "server"), ar -> {
    if (ar.succeeded() && ar.result() != null) {

        // Retrieve the service reference
        ServiceReference reference =
            discovery.getReference(ar.result());
        // Retrieve the service as HttpClient. This allows to
            make a http request to the server
        //The client created already has server's address and
            port to make the request to, thanks to the
            ServiceReference object
        HttpClient client = reference.getAs(HttpClient.class);

        System.out.println("I have retrieved the reference!");
        //path of Server get route
        client.getNow("/path1/path2/variable", response -> {

            // ...

```

```

        // Dont' forget to release the service
        reference.release();

    });
}
else{
    System.out.println("Couldn't fetch");
}
});

```

7.4 Aggiungere un nuovo Microservizio al Recommender System

Per aggiungere un nuovo microservizio al sistema, basta estendere la classe *BaseMicroService*. È sufficiente usare i metodi *protected* della classe con l'operazione *compose* per eseguire con ordine prima la creazione del server e poi la pubblicazione sul service discovery. Nell'esempio un *GreetingsMicroservice* con una GET request: specificando il nome nel path, ritorna una stringa con un saluto.

```

package eu.connecare.rs.microservices;

import io.vertx.core.Vertx;
import io.vertx.ext.web.Router;

public class GreetingsMicroservice extends BaseMicroService {

    @Override
    public void start(){
        super.start();
        Router router = Router.router(vertx);

        router.get("/greetings/:userName").handler(routingContext->{
            routingContext.response().setStatusCode(200)

```

```

        .end("Hi "+routingContext.pathParam("userName"));
    });

    createAPIServer(router)
        .compose(event -> publishHTTPEndPoint("greetings_service",
            "localhost",6578,"/greetings"));
    }
    @Override
    protected int getPort() {
        return 6578;
    }
}

```

Una volta creata la classe, basta fare il deploy del nuovo verticle nella classe *ApplicationDeployer.java*, e fare una get request al gateway con l'url

`http://localhost:8080/rs/greetings/{scrivi_qui_il_tuo_nome}`

7.5 Swagger

Creazione delle API Una volta registatosi su Swagger, creare un API premendo su *Create New*.

Il menu che compare contiene varie opzioni. Se si è nuovi allo strumento consiglio di scegliere come *template* il pet example, dato che contiene numerose API su cui studiare la sintassi del linguaggio.

Esempio: Put Request di un Prescription L'entry point da cui partire è la keyword *paths*. Sotto questa keyword, indentando di uno spazio¹² vanno scritti gli url delle risorse¹³ con cui eseguire le chiamate http.

```
paths:
  /rs/prescriptions/{prescrUUID}:
  .
  .
  .
  /rs/patients/details/{patientUUID}:
```

Sotto gli url delle risorse si possono già inserire i dettagli di design, specificando innanzitutto i parametri:

- **Parametri del percorso.** Nell'url della chiamata http è possibile inserire dei parametri che verranno usati per eseguire query o altre operazioni da parte del microservizio. Per specificare che è un parametro del percorso basta utilizzare la keyword *in*.
-

```
parameters:
- in: path
  name: prescrUUID
  schema:
    type: string
    format: uuid
    example: d290f1ee-6c54-4b01-90e6-d701748f0851
  required: true
  description: uuid of the prescription
```

- **Parametri come variabili di query.** Come nel caso precedente, la keyword da utilizzare è *query*, inserita sempre nella stessa entry *in*.
-

¹² In JSONSchema bisogna sempre indentare di uno spazio mano a mano che si inseriscono nuovi elementi

¹³ Vedere Figura n.1, "Esempio del design di una chiamata http PUT per modificare una prescription"

```
parameters:
  - in: query
    name: prescrUUID
    schema:
      type: string
      format: uuid
      example: d290f1ee-6c54-4b01-90e6-d701748f0851
    required: true
    description: uuid of the prescription
```

- **Name.** Name è il nome del parametro: nel processare la richiesta http, qualunque valore passato ai parametri verrà memorizzato in una variabile avente il nome dato.
- **Schema.** Lo schema viene usato per fornire tipo e formato del parametro. In generale, uno schema corrisponde al modello di un qualche oggetto. Ci sono vari tipi di formato e tipo che si possono usare per descrivere in modo esaustivo il parametro. La descrizione è sempre un elemento opzionale.
- **RequestBody.** Nel caso di put, delete o post si può anche specificare una request body. Il *body* viene innanzitutto descritto nel suo tipo, ovvero se è un json o un xml,

```
requestBody:
  content:
    application/json:
      schema:
        $ref: '#/components/schemas/Prescription'
```

con schema poi si può definire il modello dell'oggetto che deve essere usato. Se è un oggetto complesso, si può utilizzare un riferimento all'interno dell'editor per alleggerire l'API, spiegato in seguito.

- **Responses.** In questa parte vengono definiti i tipi di risposta che deve dare, fornendo il modello (o *schema*) degli oggetti da restituire. In questa sezione

si possono definire tutti i tipi di *response* inserendo i dettagli sotto allo status code, fornendo un esempio di ciò che viene inviato al client nel caso di una risorsa creata, di una semplice *get request* o anche per gli errori.

```
responses:
  '200':
    $ref: '#/components/responses/SuccessfulRequest'
  '400':
    description: prescription was not found
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/PrescriptionError'
  '403':
    $ref: '#/components/responses/Unauthorized'
  '404':
    $ref: '#/components/responses/NotFound'
```

Esempio: creazione di uno schema Come si è visto nel paragrafo precedente, è possibile creare i modelli degli oggetti delle API. È buona pratica creare una sezione nel file, *components*. Sotto *components* si possono specificare delle sottosezioni, in questo caso *responses* e *schemas*

```
components:
  responses:
    .
    .
    .
  schemas:
    .
    .
    .
    .
```

Tutti i modelli definiti sotto response o schemas possono essere utilizzati nelle altre parti dell'editor con la sintassi:

```
schema:
  $ref: '#/components/schemas/Error403'
```

La prima parte è la keyword e la seconda l' "url" dello schema: *Error403* si trova sotto *schemas* che è sotto *components*. Nel caso della definizione del modello della prescription, basta far riferimento alla seguente figura:

```
Accomplishment:
  type: object
  required:
    - uuid
    - patient
    - date
    - measurements
  properties:
    uuid:
      type: string
      format: uuid
      example: d290f1ee-6c54-4b01-90e6-d701748f0851
    patient:
      type: string
      format: uuid
      example: d290f1ee-6c54-4b01-90e6-d701748f0851
    date:
      $ref: '#/components/schemas/Date'
    measurements:
      type: array
      items:
        $ref: '#/components/schemas/Measurement'
```

I dettagli sono i seguenti, in ordine:

1. **Tipo.** Bisogna specificare il tipo dell'oggetto. Se non si tratta di un tipo primitivo, è obbligatorio specificare **object**.
2. **Required.** Vanno elencati come in figura i parametri che sono obbligatori, ovvero se manca qualcuno dei parametri elencati ci dovrà essere un errore
3. **Properties.** Properties fornisce i dettagli dei parametri. Se sono primitivi, si fa come nel caso della variabile di un path, altrimenti se sono oggetti si può definire un nuovo schema oppure utilizzare il riferimento

Riferimenti bibliografici

- [1] <https://www.nginx.com/blog/tag/microservices/>
- [2] <https://swagger.io/>
- [3] Alessandro Maretti, Sofia Rosetti *REST Best Practices*, 2018
- [4] <https://vertx.io/>
- [5] Alessandro Ricci, *Asynchronous Programming*, 2016/2017