Realizziamo il microservizio degli accounts

Le dipendenze inserite sono:

Spring WEB – Importantissima per realizzare le operazioni REST

H2 Database SQL

Spring Data JPA

Spring Boot Actuator – Contiene strumenti di monitoraggio di più servizi

Spring Boot Dev Tools – Strumenti base per aggiornamenti interni come riavii veloci (solo per la versione locale in fase di sviluppo e non anche per le distribuzioni finali)

Lombok – Per evitare codice boilerplate

Validation – Per controllo validazione dati in ingresso

Introdurre nelle risorse il file schema.sql con il seguente contenuto

CREATE TABLE IF NOT EXISTS `customer` (

`customer\_id` int AUTO\_INCREMENT PRIMARY KEY,

`name` varchar(100) NOT NULL,

`email` varchar(100) NOT NULL,

`mobile\_number` varchar(20) NOT NULL,

`created\_at` date NOT NULL,

`created\_by` varchar(20) NOT NULL,

`updated\_at` date DEFAULT NULL,

`updated\_by` varchar(20) DEFAULT NULL

);

CREATE TABLE IF NOT EXISTS `accounts` (

`customer\_id` int NOT NULL,

`account\_number` int AUTO\_INCREMENT PRIMARY KEY,

`account\_type` varchar(100) NOT NULL,

`branch\_address` varchar(200) NOT NULL,

`created\_at` date NOT NULL,

`created\_by` varchar(20) NOT NULL,

`updated\_at` date DEFAULT NULL,

`updated\_by` varchar(20) DEFAULT NULL

);

Poiché in entrambe le tabelle ci sono quattro campi che raccolgono metadata, ovvero info sulla creazione del record, realizziamo una tabella contenente solo questo tipo di dati, la BaseEntity

I campi tipo created\_at nel database diventano nella forma createdAt ed è garantito il sincronismo dei campi col database

Introduciamo l’annotazione @MappedSuperclass la quale trasforma la classe seguente in una super classe nei confronti di tutte le classi in cui questa verrà importata. In questa maniera le sottoclassi accederanno ai campi metadata

Inoltre inseriamo l’annotazione @Column(updatable = false) per le colonne di creazione perché non vogliamo aggiornamenti dei campi nel corso dell’esecuzione del programma ma vogliamo che in quei campi metadata rimanga il valore al momento della creazione.

Per le colonne di aggiornamento metteremo @Column(insertable = false) per dire al framework di non inserire o popolare quei campi al momento dell’inserimento di un nuovo record.

In tutto questo, non abbiamo bisogno di inserire nomi arbitrari di colonne in quanto, allo stato attuale, corrispondono lato spring e lato database

Fatti anche gli accounts, realizziamo le classi repository

Realizziamo la classe DTO riguardanti clienti e conti (per adesso non aggregati)

Oltre a questi DTO legati alle classi database, inseriamo classi dto riguardanti la richiesta di salvataggio dati nel sistema

Per questo le chiamiamo responseDto e ErrorResponseDto

Realizziamo, sul controller, la funzione createAccount

Realizziamo prima un package di costanti

Realizziamo l’interfaccia e classi di servizio

Adesso il package mapper per convertire le classi entità in dto e viceversa

Nella classe di servizio, oltre a realizzare un nuovo cliente, gli assegniamo dei conti bancari di cui il numero conto è un valore casuale di cifre numeriche

Adesso il package exceptions per gestire eventuali errori di omonimia dei clienti

Il campo univoco per capire se un cliente sia stato già immesso è il numero di cellulare

Dobbiamo quindi vedere se il cellulare del contatto che stiamo inserendo, non sia già in utilizzo da un altro cliente che diventerebbe cliente reale

Quindi, scriviamo la funzione in CustomerRepository che cercherà l’eventuale (infatti non è detto che esista quindi usiamo la parola optional) clinte col numero di cellulare

Optional<Customer>findByMobileNumber(String mobileNumber);

Ovviamente in questo caso, grazie alle funzioni JPA, non c’è bisogno di un ulteriore implementazione

GlobalExceptionHandlingLogic per dare una logica comune di intervento alle eccezioni quando si verificano

L’annotazione @ControllerAdvice sorveglia se capita un caso di eccezione in ogni punto del progetto

Ricordiamoci di dare un valore default a created\_at e created\_by in tutte le entità in cui vengono richiamati

Sulla validazione:

Le annotazioni di validazione (NotEmpty, ecc..) vanno fatte sui campi dto

Va messa l’annotazione @Validated nel controller per poter applicare le regole sui singoli campi

Sempre sul controller

Dobbiamo mettere l’annotazione @Valid dove utilizziamo @requestBody e prima di esso (@Valid @RequestBody)

Mentre sul fetchAccount, dopo la @requestParam, ripetiamo l’annotazione per il cellulare

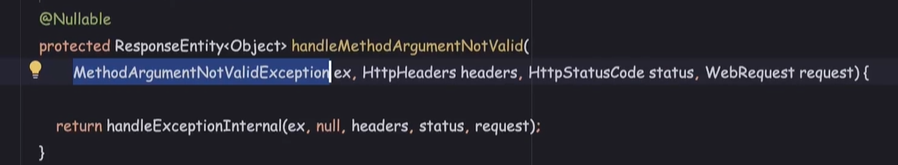
@Pattern(regexp = "(^$|[0-9]{10})", message = "Il numero cellulare deve essere di 10 cifre!")

E anche nel deleteAccount

Una volta messe le annotazioni dobbiamo risolvere il seguente problema: Il framework sa come rispondere alle convalide che falliscono grazie ai messages immessi, ma non sa come inserire questi messaggi all’interno delle risposte

Per risolvere il problema dobbiamo andare nella GlobalExceptionHandler e estendere la ResponseEntityExceptionHandler

Dentro questa ulteriore superclasse esiste il metodo handleMethodArgumentNotValid



Auditing

Vediamo il funzionamento degli audit

Questo meccanismo dovrebbe aggiornare automaticamente i campi di: created\_at, created\_by, updated\_at. Updated\_by

Anche questo tipo di aggiornamenti possono essere controllati automaticamente dalla Spring data JPA

Il primo passo da fare è assegnare ulteriori notazioni ai campi della classe che gestisce questi metadata (BaseEntity), nell’ordine della seguente tabella

|  |  |
| --- | --- |
| **Campo** | **Annotazione da aggiungere** |
| createdAt | CreatedDate |
| createdBy | CreatedBy |
| updatedAt | LastModifiedDate |
| updatedBy | LastModifiedBy |

Con queste annotazioni automatizziamo il contenuto dei nostri campi metadata

Però dobbiamo specificare cosa devono inserire: Chi realizza la creazione o l’aggiornamento? Si deve inserire l’ora dal server o dal sistema locale?

Dobbiamo quindi configurare un file apposito di impostazioni

Realizziamo il nuovo package “audit” e, dentro, la classe “AuditAwareImpl”

Questa classe implementa la superclasse AuditAware<String>

selezionare getCurrentAuditor

Questo metodo, che parte Optional.empty, stabilisce “l’autore” degli aggiornamenti

Lo aggiorniamo con Optional.of(“ACCOUNTS\_MS”)

Quando si affronterà il Spring Security, questo ruolo sarà affidato all’utente attualmente loggato

Stabiliamo anche l’annotazione @Component(“auditAwareImpl”) ovviamente sopra public class AuditAwareImpl

Questo è necessario per poi tornare alla classe che gestisce i campi metadata (BaseEntity) e dirgli, tramite @EntityListeners(AuditingEntityListener.class) dove AuditingEntityListener è la componente auditing di JPA.

Successivamente, andare nella classe principale di avvio dell’applicazione (AccountsApplication nel nostro caso) ed inserire l’annotazione @EnableJpaAuditing(auditorAwareRef=”auditAwareImpl”) dove auditAwareImpl è il nome del component nella classe AuditAwareImpl

Funziona

Adesso vediamo la documentazione con OpenAPI

Una volta installata la dipndenza sul pom, vediamo il risultato con

localhost:8080/swagger-ui/index.html

e ci darà la pagina html con la documentazione del nostro servizio

Bisogna usare una versione apposita di openai

Ecco la dipendenza giusta (e non ui)

<https://mvnrepository.com/artifact/org.springdoc/springdoc-openapi-starter-webmvc-ui/2.6.0>

A proposito del problema project in pom

<https://github.com/spring-projects/spring-framework/issues/32219>

Sostanzialmente dice di sostituire <https://maven.apache.org/xsd/maven-4.0.0.xsd> con <http://maven.apache.org/xsd/maven-4.0.0.xsd> togliendo la s finale in https

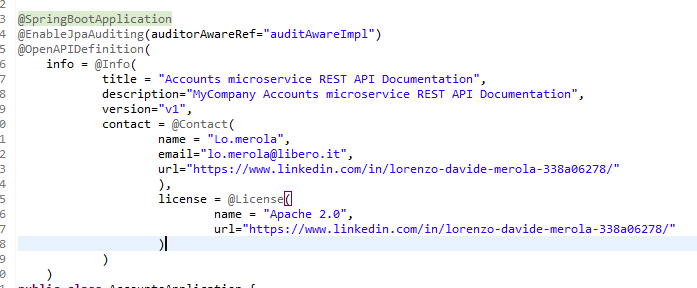
Tornando alla documentazione openapi.

Data la versione base, adesso vediamo possibili miglioramenti.

La prima cosa che potremmo aggiornare è l’intestazione che, per default, mostra “OpenAPI definition”

Primo passo. Andare alla classe di avvio (AccountsApplication) e introdurre la dipendenza OpenAPIDefinition

Ecco un esempio di applicazione



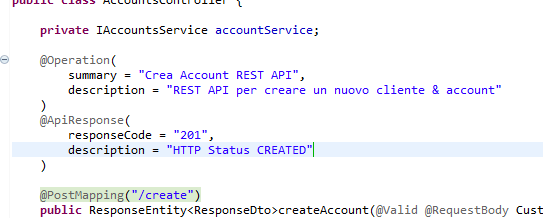
Possiamo aggiornare anche la sezione controller

Aggiungiamo la notazione @Tag, in controller, nella seguente maniera

A close-up of a computer screen

Description automatically generated

Poi possiamo aggiungere dettagli alle singole operazioni con il tag Operation e ApiResponse



A screenshot of a computer

Description automatically generated

Possiamo applicare le notazioni aggiuntive sulla guida anche alle singole classi dto

Sia a livello di classe, che a livello di campo

A screenshot of a computer

Description automatically generated

La stessa cosa anche per la classe accounts (conti)

A screenshot of a computer program

Description automatically generated

Anche per la classe risposta

A screenshot of a computer program

Description automatically generated

… e anche per la classe errore

A screenshot of a computer screen

Description automatically generated

Nella documentazione finale non compare l’oggetto errorResponse. Questo perché l’oggetto errorResponse viene invocato dalla GlobalExceptionHandler che resta esclusa da questa documentazione.

Ne segue che, per accedere alla documentazione dell’oggetto response, dobbiamo aggiungere qualche istruzione in più quando si rientra nelle situazioni di errore.

Così, per esempio, nel caso dell’update abbiamo nella situazione di errore 500, avremo:

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

Adesso abbiamo 2 servizi da fare simili a questo primo accounts che sarebbero carte (cards) e prestiti (loans)

Cominciamo dalle carte e cerchiamo di capire le entità da mettere guardando il github

Facciamo le fasi preliminari

1. Realizziamo il progetto con start.spring.io

A screenshot of a computer

Description automatically generated

Mettiamo le dipendenze iniziali che sarebbero:

Spring web (per la parte rest response)

H2 database

Spring Data JPA (support a h2 database)

Common Dev Tools (strumenti per un veloce riavvio in locale)

Lombok

Spring Boot Actuator – Contiene strumenti di monitoraggio di più servizi

Validation per i controlli di validazione

1. Avviato il progetto realizziamo 2 classi essenziali:

BaseEntity che conterrà la gestione dei metadati

ResponseDto – classe gestione messaggi errore

Quindi realizziamo i package entity e dto

Procedura per realizzare BaseEntity

* Realizzare i seguenti campi

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Column** | **Audit** |
| createdAt | LocalDateTime | updatable = false | CreatedDate |
| createdBy | String | updatable = false | CreatedBy |
| updatedAt | LocalDateTime | insertable = false | LastModifiedDate |
| updatedBy | String | Ínsertable = false | LastModifiedBy |

* Aggiungere le seguenti annotazioni per intestazione

|  |  |
| --- | --- |
| **Annotazione** | **Descrizione** |
| MappedSuperClass | Designa una classe le cui informazioni di mappatura vengono applicate alle entità che ereditano da essa. |
| EntityListeners |  |
| Gette, Setter, ToString | Lombok |