



Progetto del corso “Introduzione alla programmazione per il web”

Documento relativo alle scelte implementative

Obiettivo: Progettare un sistema di gestione e condivisione di liste della spesa da esporre via WEB e fruibile attraverso dispositivi desktop e, in modalità responsive, anche via mobile.

Il sistema prevede di essere utilizzato in tre modalità:

Utente anonimo: utente non salvato nel database del sistema, ma che può effettuare tutte le operazioni dell'utente registrato ad esclusione delle funzionalità di condivisione;

Utente registrato: l'utente ha inserito un proprio username e password e può condividere la lista con altri utenti del sistema;

Amministratore: Utente che gestisce il sistema creando oggetti, categorie di liste della spesa (supermercato, ferramenta, farmacia, ecc...) che altri utenti possono usare.

Progetto realizzato da:

Berisha Gentiana

Degiacomi Simone

Facchini Rublev Viatcheslav

Faieta Alessio

Architettura

La webapp è composta da due componenti principali: un server, che espone delle API REST, e un client, di tipo Single Page Application (SPA). Il client comunica con il server utilizzando il protocollo HTTP e trasferendo oggetti nel formato JSON. Oltre ad esporre gli endpoint per lo scambio dei dati, il server si occupa anche di servire i file statici (script JavaScript e file CSS) che compongono il client.

Perchè una single-page application?

Un client SPA è un'applicazione eseguita all'interno del browser e che cambia in modo dinamico il DOM della pagina web senza la necessità di ricaricarla. Abbiamo deciso di realizzare il progetto con questa tecnica perché essa offre due vantaggi principali:

- **Migliore user experience:** L'utilizzo delle webapp si sposta sempre di più da computer a smartphone. Oltre a questo gli utenti si aspettano tempi di risposta sempre più brevi. La caratteristica di non aggiornare la pagina fornisce un'esperienza più fluida e naturale, simile a quella ottenuta da programmi o applicazioni native.
- **Migliore separazione del client dal server:** Nelle SPA vi è una naturale divisione tra le funzioni del client e quelle del server. Questo implica che le funzioni del server, una volta create, possono essere utilizzati da una varietà di client diversi. Per esempio, se in futuro volessimo realizzare un nuovo client per il nostro servizio (per esempio un'app nativa per smartphone) sarà possibile riutilizzare lo stesso server senza doverlo modificare.

Client SPA

I client SPA vengono realizzati attraverso script JavaScript che si occupano di richiedere al server i dati da mostrare. Per semplificare l'organizzazione e per non dover scrivere troppo codice di supporto, abbiamo scelto di adottare uno dei tanti framework disponibili: quello che abbiamo scelto è [Angular](#), versione 6.

Perchè Angular?

Le funzioni offerte da questo framework sono molte, ma quelle che hanno catturato la nostra attenzione sono:

- Supporto a [TypeScript](#): il tool ng-cli permette di utilizzare in modo semplice e out-of-the-box il linguaggio di programmazione TypeScript, che viene poi transpilato in JavaScript. TypeScript è un linguaggio open source creato da Microsoft che aggiunge a JavaScript molti dei vantaggi dei linguaggi tipizzati;
- Divisione dell'app in **componenti**: Le parti grafiche della webapp vengono divise in componenti, i quali sono riutilizzabili e aiutano a dividere il codice per mantenerlo

semplice anche quando le funzioni da implementare sono molte. Ogni componente grafico corrisponde ad una cartella contenente tre file: il controller (una classe di TypeScript), il layout (scritto in HTML) e lo stile (scritto in CSS);

- **[Dependency Injection](#)**: La DI è una tecnica che permette di fornire ai componenti classi o oggetti istanziati da altri componenti dell'applicazione. La DI è una tecnica di "Inversion of control" che aiuta a seguire il "Open/Close principle".

Stile grafico

La grafica dell'applicazione è stata realizzata da noi con Inkscape (alternativa opensource ad Illustrator). Prima di cominciare a programmare abbiamo creato dei mockup delle varie schermate dell'applicazione; queste ci sono servite da guida per avere uno stile grafico consistente nell'applicazione. L'implementazione in HTML si basa su [Bootstrap](#), che abbiamo personalizzato. I vantaggi di utilizzare Bootstrap come base consistono nell'utilizzo di classe CSS già create che aiutano a dividere il contenuto rendendolo responsive, senza dover scrivere regole che utilizzano i media query.

Server REST

Il server è stato implementato utilizzando il linguaggio di programmazione [Kotlin](#) e il framework [Spring Boot](#).

Perché Kotlin?

Il compilatore Kotlin permette di generare diversi output, tra cui bytecode per la JVM. Come per le applicazioni Java, è possibile generare un archivio .jar oppure .war.

Kotlin estende le funzionalità di Java, rendendolo più conciso e sicuro; si tratta di vantaggi che fanno sì che i programmatori commettano meno errori durante lo sviluppo, producendo quindi software con un tasso di difettosità più basso ed in meno tempo. Kotlin raggiunge questi obiettivi fornendo:

- Costrutti per evitare i NullPointerException: Un valore null inatteso è la ragione più frequente di crash del software. Si tratta di un errore così diffuso che l'inventore del null si è [pubblicamente scusato per la sua creazione](#). Kotlin è null-safe. Questo significa che le variabili non possono ricevere valore null, se non esplicitamente specificato. Quando si specifica che una variabile è null, Kotlin forzerà il programmatore a controllare che la variabile non sia null prima permettergli di fare accesso, pena un errore di compilazione.
- Vari costrutti per rendere il codice meno verboso: Java è spesso criticato per la sua eccessiva verbosità che rende il suo codice difficile da seguire. Kotlin aiuta in questo fornendo strumenti quali inferenza di tipo, smart casts, delegazioni di proprietà, estensioni di funzioni, object declarations e data-classes.
- Vi sono anche [altri strumenti utili per le operazioni più comuni](#)

Perchè Spring Framework?

Abbiamo deciso di utilizzare alcuni componenti del framework Spring per semplificare lo sviluppo. Una delle funzionalità più comode portate dal framework è, ancora una volta, la Dependency Injection. Gli altri componenti di Spring che abbiamo utilizzato sono:

- Spring Security: ci permette di configurare le regole di autenticazione e autorizzazione direttamente nel codice o utilizzando annotazioni per classi e metodi.
- Spring WebSocket e STOMP: Componenti che ci permettono di utilizzare in modo semplice all'interno delle servlet i protocolli WebSocket e STOMP. Il motivo per l'utilizzo di questi protocolli è spiegato più in dettaglio nelle sezioni successive del documento.
- Componenti per il web: Annotazioni per associare metodi a URL e classi per gestire le richieste HTTP.
- Spring JPA: Anche questo componente verrà spiegato in una sezione dedicata.

La configurazione di Spring è stata gestita mediante Spring Boot, un configuratore automatico dei componenti di Spring.

Perchè Hibernate e Spring JPA?

Per gestire il layer di persistenza abbiamo utilizzato l'ORM Hibernate e il componente Spring JPA.

Hibernate è un ORM, uno strumento che ci permette di utilizzare un database relazionale senza dover scrivere manualmente le query o il codice che mappa le tabelle e le righe di esse in classi e oggetti.

Hibernate è in grado di generare query per inserire, cercare, eliminare e aggiornare istanze di entità. Grazie al componente Spring JPA possiamo fruire di query più complesse dichiarando delle interfacce: scegliendo opportunamente il nome dei metodi delle interfacce, Spring si occuperà di generare le implementazioni reali con le relative query. Nei pochi casi in cui Spring non è in grado di generare una query, possiamo specificarla noi utilizzando un'apposita annotazione.

Tutte le query sono automaticamente protette da attacchi di SQL injection, cosa utilissima per una web app come la nostra.

Persistenza

Hibernate permette di cambiare database relazionale senza dover riscrivere il codice dell'applicazione che lo utilizza; è sufficiente specificare il driver da utilizzare e i parametri di connessione. Nel nostro caso abbiamo deciso di utilizzare il database H2 in memoria. Questo database ci permette di avviare l'applicazione in tempi molto brevi e senza dover apportare modifiche alle tabelle nel caso di modifiche alla struttura di esse durante lo sviluppo (visto che le tabelle vengono ricreate ad ogni avvio). Visto che il database perde tutti i suoi dati ad ogni

avvio, abbiamo realizzato una classe che si occupa di importare dei dati di test durante l'avvio, leggendo dei file csv.

Per portare in produzione questo servizio ed evitare la perdita dei dati ad ogni avvio, è sufficiente scegliere un database persistente (es: mariadb) e specificare il nome del driver adatto con qualche parametro di connessione (host, username e password).

Funzionalità

Notifiche

Nella nostra applicazione web è di fondamentale importanza tenere gli utenti aggiornati sulle modifiche fatte dai propri amici. Per fare ciò abbiamo implementato delle notifiche che potessero raggiungere l'utente anche quando non ha il sito web aperto nel proprio browser, utilizzando le [Push Notification](#).

Per utilizzare questa API il client chiede all'utente il permesso di mostrare le notifiche (chiamando un'API offerta dal browser). Nel caso in cui l'utente accettasse, il browser fornisce al client JavaScript un token e un url univoci. Il client si occuperà poi di inviare questi dati al server.

D'ora in poi, ogni volta che il server vuole notificare un client, dovrà solamente inviare all'URL fornito dal client la notifica contenente il token univoco. I server della casa produttrice del browser si occuperanno poi di notificare il dispositivo dell'utente.

Questo procedimento è sufficiente per supportare il browser Mozilla Firefox, ma non per Google Chrome. Per supportare quest'ultimo, è necessario o utilizzare i servizi di Firebase o [generare delle chiavi VAPID](#). Per utilizzare un unico metodo di invio delle notifiche per tutti i browser, abbiamo scelto la seconda opzione.

Dopo aver generato una coppia di chiavi VAPID, dobbiamo informare il browser che vogliamo utilizzare questa funzione facoltativa mentre chiediamo il permesso di mostrare notifiche all'utente e criptare le notifiche, prima di inviarle ai server dei browser, con la nostra chiave privata.

Geolocalizzazione per notificare i negozi di interesse vicini

La nostra webapp fornisce anche un sistema di suggerimenti di negozi vicini che vendono oggetti che un utente ha attualmente in una lista. Per fare ciò il client richiede all'utente il permesso della [geolocalizzazione](#) e invia al server, ogni volta che il dispositivo fa uno spostamento significativo (centinaia di metri), le proprie coordinate. Il server è a conoscenza delle categorie di liste della spesa che l'utente gestisce e quindi può cercare negozi vicini di categorie simili. Questo perché gli amministratori possono associare categorie di liste della spesa a categorie del servizio di mappe che abbiamo scelto, nel nostro caso [Foursquare](#).

Sfortunatamente non ci è stato possibile implementare questa feature in modo che funzionasse anche quando la webapp è chiusa: è necessario avere la pagina web aperta sul proprio

dispositivo mobile per rimanere aggiornati sui negozi vicini. Questo perché, pur utilizzando i [Service Worker](#) e i manifest per la creazione di [PWA](#), i browser non forniscono ancora delle API per accedere alla posizione del dispositivo in background. Se si volesse implementare questa funzione con le tecnologie attuali sarebbe necessario implementare un'applicazione nativa.

Sincronizzazione in tempo reale e chat

Come già anticipato nella sezione delle notifiche, la nostra app mantiene gli utenti costantemente aggiornati.

Il client scarica le liste della spesa dell'utente all'apertura della webapp nel browser. Per mantenere aggiornate le liste in tempo reale abbiamo deciso di utilizzare i [WebSocket](#). In pratica il client apre un canale web socket all'avvio e tramite il protocollo [STOMP](#) si iscrive agli aggiornamenti dei topic di cui gli interessa seguire gli aggiornamenti.

Il nostro servizio utilizza i WebSocket solo per garantire la sincronizzazione in tempo reale e quindi il servizio può essere utilizzato a pieno anche nel caso in cui una connessione WebSocket non possa essere creata.

Funzionalità “Prova ora”

Nella landing page è presente un pulsante che permette agli utenti non registrati di accedere ad un'unica lista della spesa. Quando un utente anonimo aggiunge prodotti a questa lista demo, i dati dei prodotti vengono salvati nel [local storage](#) del browser. Ovviamente, a differenza degli utenti autenticati, la sua lista non sarà sincronizzata sul server.

Internazionalizzazione

L'internazionalizzazione consiste nel rendere un servizio disponibile a vari utenti di diverse lingue e culture. Nel nostro caso abbiamo deciso di rendere il servizio disponibile nella lingua italiana (it-IT) e inglese (en-US). Dato che nella nostra webapp vengono generati contenuti sia lato client che lato server (notifiche e email), abbiamo dovuto utilizzare due sistemi di internazionalizzazione diversi.

Client

Angular fornisce di default degli [strumenti per semplificare l'implementazione](#), attraverso dei semplici passaggi: in pratica bisogna modificare i template dei componenti e aggiungere la keyword `i18n` ai tag HTML che vogliamo tradurre. Chiamando poi il tool `ng-cli` possiamo generare dei file xml che noi modificheremo in modo da aggiungere le traduzioni dei testi. Il problema di questa funzionalità consiste nel fatto che poi dovremo generare due diverse webapp (in pratica script diversi) da servire al client. Questo avrebbe reso il processo di sviluppo più complesso (è possibile debuggare solamente una lingua alla volta) e per questo abbiamo deciso di utilizzare una libreria alternativa che permette di passare da una lingua all'altra a runtime. Questa libreria si chiama [ngx-translate](#). L'utilizzo di questa libreria è diverso dal metodo ufficiale proposto da Angular, ma è comunque semplice: anziché scrivere i testi all'interno dei template dei nostri componenti, dobbiamo inserire delle chiavi che identificano in

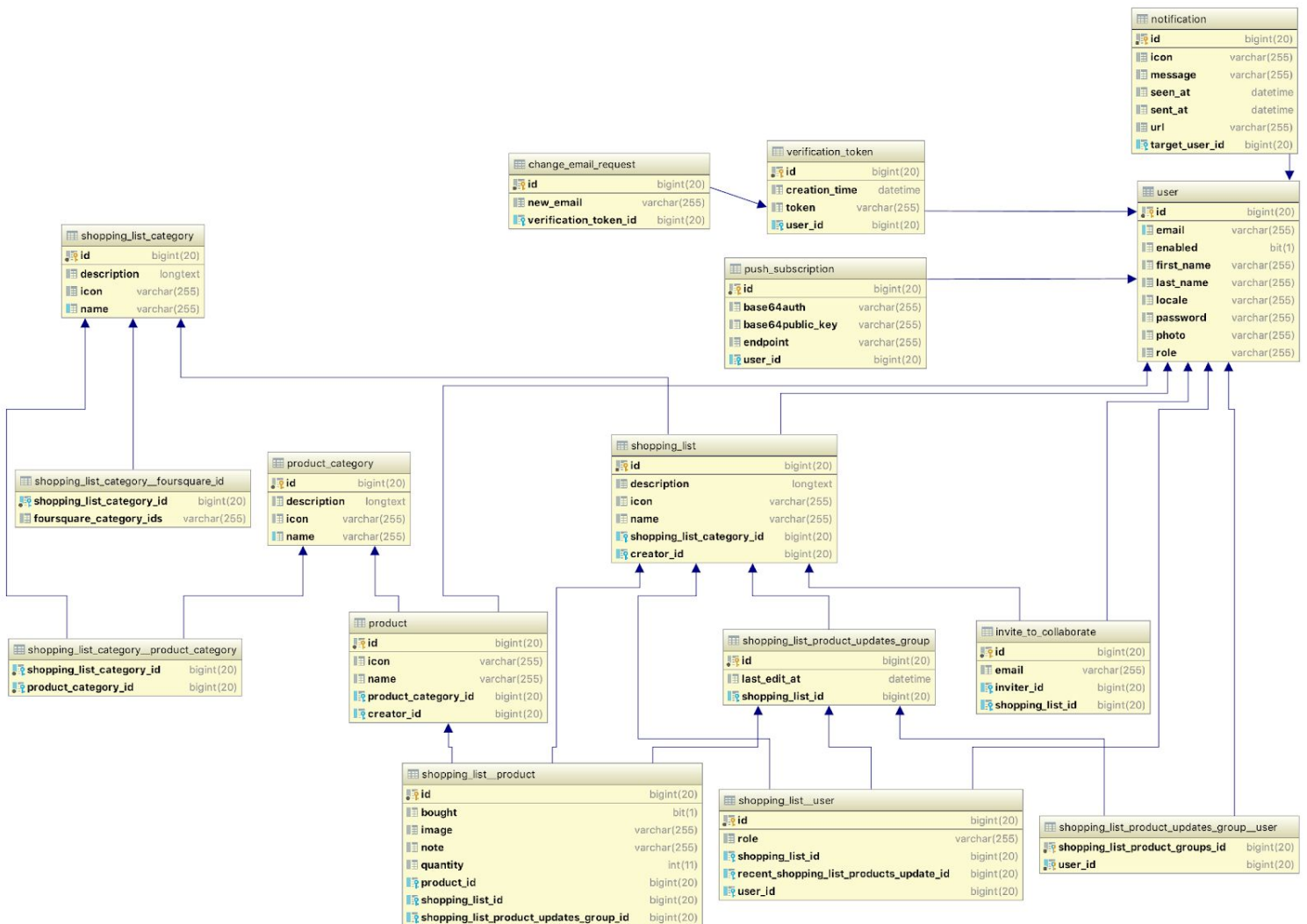
modo univoco i testi. In file JSON separati, uno per ogni lingua supportata, andremo poi a inserire i testi tradotti.

Server

Per quanto riguarda il lato server abbiamo optato per diversi (uno per ogni locale) file JSON (come per il client) che ci occupiamo di caricare in memoria all'avvio. Quando generiamo delle email inseriamo nelle stringhe delle chiavi, delimitate dai caratteri '{{' e '}}'. Queste sottostringhe vengono poi sostituite con i valori corretti decisi dai contenuti dei file JSON.

Diagramma ER

Di seguito è presente il diagramma ER del database dell'applicazione:



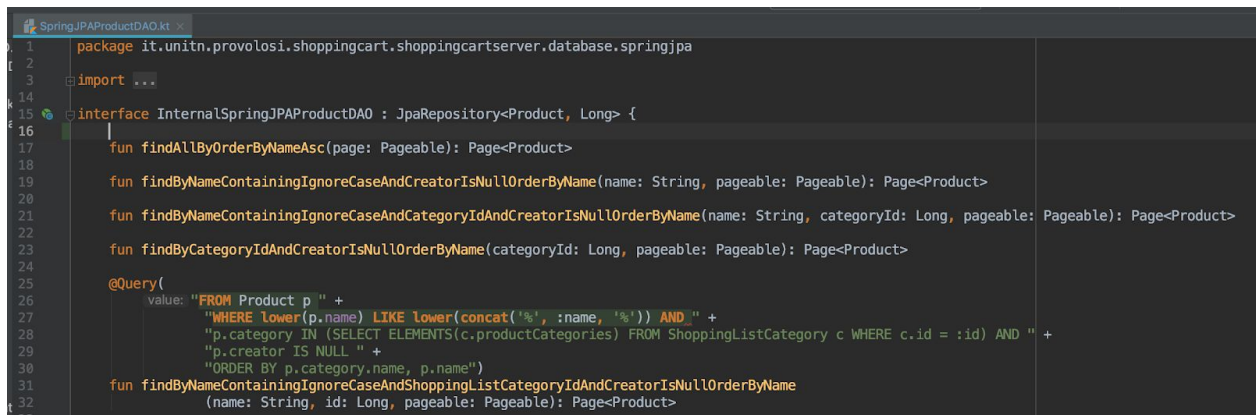
Classi principali

DAO

Per ogni entità (Product, ProductCategory, ShoppingList, ecc...) sono presenti due interfacce e una classe. Prendendo in considerazione l'entità Product, esse sono:

- ProductDAO: Interfaccia che espone i metodi per creare, modificare, leggere, eliminare e cercare un prodotto. Questa interfaccia non utilizza nessuna classe specifica di Hibernate o SpringJPA, dipende solamente dalle classi definite dalla nostra applicazione;
- InternalSpringJPAProductDAO: Interfaccia che estende l'interfaccia [JpaRepository](#) di Spring. L'interfaccia di Spring espone diversi metodi che permettono le classiche operazioni CRUD, quindi i metodi definiti da noi riguardano esclusivamente le query specifiche della logica della nostra applicazione. I nomi dati a questi metodi non sono scelti liberamente ma seguono delle precise regole dettate da Spring: in questo modo verranno poi generate automaticamente le query.

Proprio nell'interfaccia dell'entità Product si verifica il caso in cui Spring non è in grado di generare alcune query che ci servono e quindi le specifichiamo noi utilizzando la notazione @Query:



```
1 package it.unitn.provolosi.shoppingcart.shoppingcartserver.database.springjpa
2
3 import ...
4
5 interface InternalSpringJPAProductDAO : JpaRepository<Product, Long> {
6
7     fun findAllByNameAsc(page: Pageable): Page<Product>
8
9     fun findByNameContainingIgnoreCaseAndCreatorIsNullOrderByName(name: String, pageable: Pageable): Page<Product>
10
11     fun findByNameContainingIgnoreCaseAndCategoryIdAndCreatorIsNullOrderByName(name: String, categoryId: Long, pageable: Pageable): Page<Product>
12
13     fun findByCategoryIdAndCreatorIsNullOrderByName(categoryId: Long, pageable: Pageable): Page<Product>
14
15     @Query(
16         value: "FROM Product p " +
17             "WHERE lower(p.name) LIKE lower(concat('%', :name, '%')) AND " +
18             "p.category IN (SELECT ELEMENTS(c.productCategories) FROM ShoppingListCategory c WHERE c.id = :id) AND " +
19             "p.creator IS NULL " +
20             "ORDER BY p.category.name, p.name")
21     fun findByNameContainingIgnoreCaseAndShoppingListCategoryIdAndCreatorIsNullOrderByName(
22         name: String, id: Long, pageable: Pageable): Page<Product>
23 }
```

- SpringJPAProductDAO: la classe contenente l'implementazione del ProductDAO. Questa classe riceve, nel suo costruttore, un'implementazione generata da Spring di InternalSpringJPAProductDAO. L'implementazione di ogni metodo consiste solo nel chiamare il metodo corretto sull'oggetto InternalSpringJPAProductDAO.

```

44  @Component
45  class SpringJPAProductDAO(
46      @Autowired
47      private val springRepository: InternalSpringJPAProductDAO
48  ) : ProductDAO {
49
50      override fun findAllByOrderByNameAsc(page: Pageable): Page<Product> = springRepository.findAllByOrderByNameAsc(page)
51
52      override fun save(product: Product) = springRepository.save(product)
53
54      override fun findByNameContainingIgnoreCaseAndCategoryIdAndCreatedByAdminOrderByName
55          (name: String, categoryId: Long, pageable: Pageable) =
56          springRepository.findByNameContainingIgnoreCaseAndCategoryIdAndCreatorIsNullOrderByName(name, categoryId, pageable)
57
58      override fun findByCategoryIdAndCreatedByAdminOrderByName(categoryId: Long, pageable: Pageable) =
59          springRepository.findByCategoryIdAndCreatorIsNullOrderByName(categoryId, pageable)

```

Come mai abbiamo scelto questo approccio? Facendo così possiamo utilizzare le funzionalità di SpringJPA senza legarci troppo ad esso: infatti, in tutta l'applicazione, le classi useranno il database solamente attraverso l'interfaccia indipendente da Spring e da Hibernate (ProductDAO).

Inoltre, dato che l'implementazione fornita da Spring è utilizzata all'interno di una classe (SpringJpaProductDAO), possiamo utilizzare la funzione di generazione delle query attraverso il nome del metodo senza dover cambiare nomi ai metodi dichiarati nell'interfaccia non dipendente (ProductDAO).

Raggruppamento delle notifiche

Quando in una lista condivisa uno (o più utenti) modificano lo stato di alcuni prodotti (la quantità, le note, l'icona o il fatto che il prodotto sia da acquistare oppure no) viene generata una notifica che raggruppa le modifiche. Questo perché è importante mantenere notificati tutti i collaboratori, ma non vogliamo nemmeno disturbarli troppo: basta immaginare il caso in cui un nostro amico ha comprato 10 prodotti e quindi va ad aggiornare il loro stato nella lista. Di certo non vogliamo ricevere 10 notifiche diverse, ma un'unica notifica.

Per implementare questa funzione di raggruppamento abbiamo creato l'entità ShoppingListProductUpdatesGroup. Di istanze di questa entità ce ne può essere al massimo una sola contemporaneamente per lista. Quando questa entità è presente essa viene riferita alla lista, alle collaborazioni degli utenti che la stanno modificando e alle relazioni che legano i prodotti all'interno di una lista.

La gestione di questa entità è compito della classe ShoppingListProductsUpdateService che espone un unico metodo collectEvent chiamato dai vari controller che gestiscono le richieste HTTP.

Il metodo collectEvent controlla se la modifica per cui è stato chiamato è una nuova modifica nella lista: in tal caso crea l'entità ShoppingListProductUpdatesGroup, altrimenti aggiorna il timestamp dell'ultima modifica e fa riferire l'entità al collaboratore o al prodotto appena modificato.

Nel mentre, ad intervalli fissi, un metodo della classe `ShoppingListProductsUpdateTask` viene invocato. Questo metodo si occupa di leggere dal database tutte le entità che rappresentano gruppi di modifiche e, nel caso in la lista relativa non sia modificata da un certo tempo (es: 1 minuto) si occupa di creare delle notifiche, inviarle e eliminare l'entità `ShoppingListProductUpdatesGroup`.

Implementazione funzione “Prova ora”

Nel client è presente un unico componente che si occupa di mostrare una lista all'utente e che fornisce i controlli per aggiungere prodotti o per modificare il loro stato. Questo componente delega il salvataggio delle modifiche alla lista (nonché il caricamento della lista) al servizio `ShoppingListService`.

`ShoppingListService` è un'interfaccia di TypeScript e abbiamo configurato Angular in modo da utilizzare un'implementazione basata su local storage nel caso in cui si stiano modificando la lista demo e un'implementazione basata su chiamate HTTP nel caso di una lista normale.