E-shopper

Progettazione Tecnica & analisi

Sommario

1 Analisi 2

1.1 descrizione applicazione 2

1.1.1 Caso d’uso: Gestione Account 3

1.1.2 Caso d’uso: Gestione Categorie, prodotti e item 3

1.1.3 Caso d’uso: Gestione Carrello della spesa 4

1.1.4 Caso d’uso: Gestione Ordini 5

1.1.5 Caso d’uso: Gestione Lista dei desideri (wishlist) 5

2 Progettazione tecnica 6

2.1 Quadro generale 6

2.1.1 E-shopper 7

2.2 SERVIZI INFRASTRUTTURALI 7

2.2.1 Microservice-demo-config 8

2.2.2 Microservice-demo-registration 8

2.2.3 Microservice-demo-security 10

2.2.4 Microservice-demo-gateway 11

2.3 Ambiente di sviluppo 12

2.3.1 Il database 13

2.3.2 Come avviare i microservizi 14

2.4 Lo sviluppo 16

# Analisi

E-shop è un’applicazione demo realizzata con uno stile architetturale a microservizi che permette l’acquisto di capi di abbigliamento femminile.

## descrizione applicazione

La navigazione del sito prevede due scenari differenti a seconda se si è registrati oppure no.

In caso di utente pubblico le azioni permesse sono:

* registrazione
* consultazione dello shop (filtri: categoria/prodotti/prezzo)
* invio di mail (al momento solo simulato).

In caso di utente registrato le azioni permesse (oltre a quelle pubbliche) sono:

* login
* setting account
* gestione di spedizione
* gestione di metodi di pagamento
* inserimento ed utilizzo di gift card
* gestione di una lista dei desideri
* inserimento di ordini e gestione di quest’ultimi

### Caso d’uso: Gestione Account

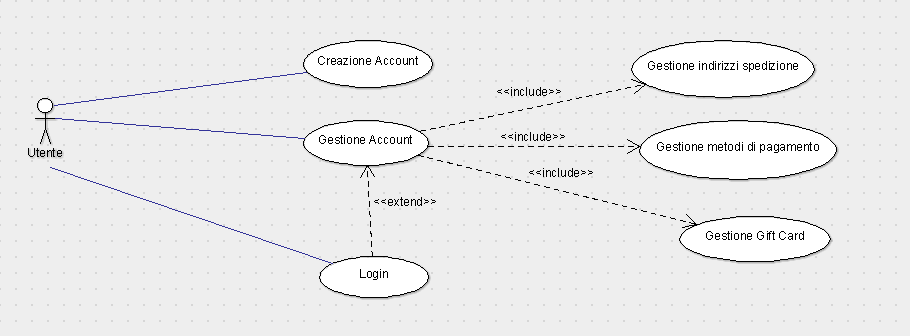
Un utente ha la possibilità di manipolare i propri dati.

Nello specifico è libero di:

* poter modificare il nome e/o la password
* gestire indirizzi di spedizione e metodi di pagamento (inserimento, modifica e cancellazione logica)
* inserire giftcard.

In ogni schermata di inserimento e modifica, oltre ai controlli di validità, sono presenti controlli sull’obbligatorietà dei campi.

In questa sezione è possibile anche disattivare un account (previa ulteriore conferma da parte dell’utente) e inserire una nuova password nel caso in cui non si ricordi quella precedente a fronte di un controllo su email dell’utente.

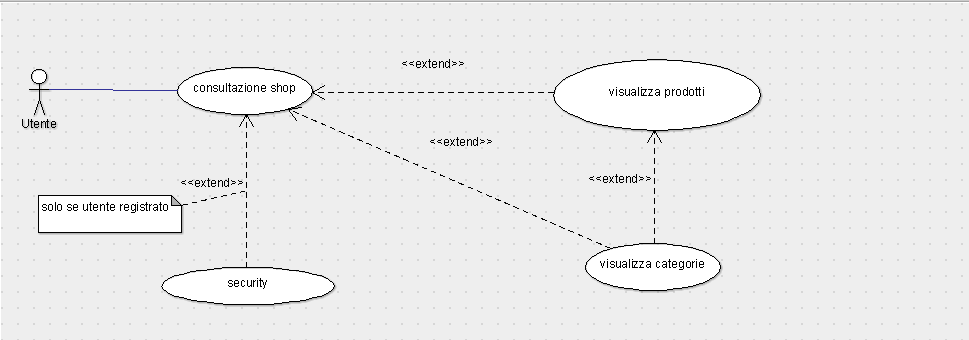


### Caso d’uso: Gestione Categorie, prodotti e item

Gli item sono raggruppati per prodotti e questi ultimi per categorie. È possibile trovare però categorie che non hanno relazioni con i prodotti, ma direttamente con gli item stessi.

Per ogni item vengono mostrate le seguenti informazioni:

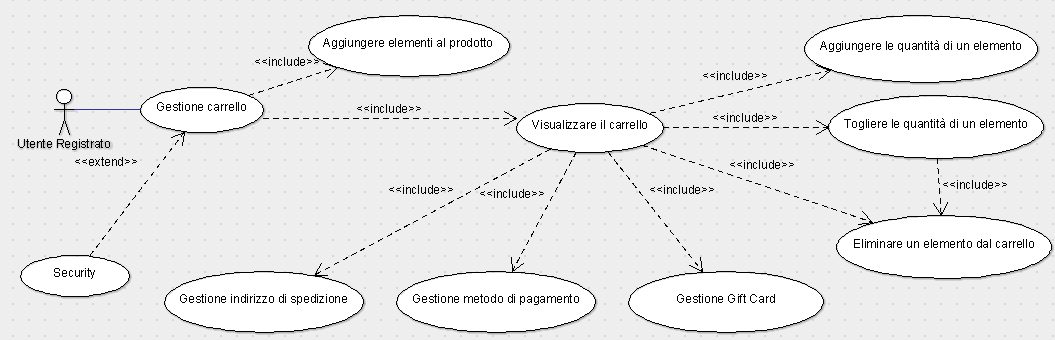
* immagine rappresentativa del prodotto
* costo
* codice univoco
* in magazzino non vi è più disponibilità
* in magazzino è rimasta una sola unità



### Caso d’uso: Gestione Carrello della spesa

Un utente registrato può aggiungere un item nel carrello della spesa in molteplici sezioni dell’applicativo: *home, shop, lista dei desideri e ordini*. Può anche decidere nella prima schermata del carello, se aumentare o diminuire le quantità da acquistare di un determinato item o se cancellarlo definitivamente dal carrello.

In fase di checkup avrà la possibilità di modificare sia l’indirizzo di spedizione che il metodo di pagamento (potrà anche decidere se utilizzare una delle gift card precedentemente inserite); entrambi sono obbligatori e non sarà possibile andare avanti con le azioni richieste fin quando non saranno stati selezionati.

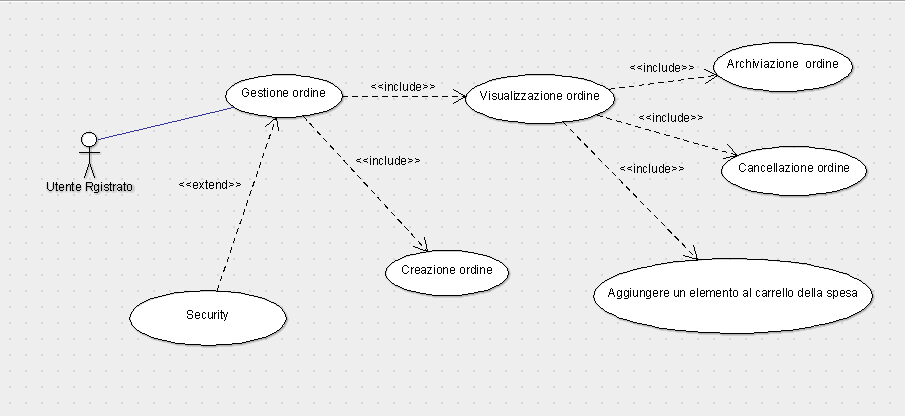


### Caso d’uso: Gestione Ordini

Una volta completato un ordine, se esso non è ancora concluso potrà cancellarlo, andando o in “order” oppure nella sezione “orders” all’interno di “account” e in seguito archiviarlo.

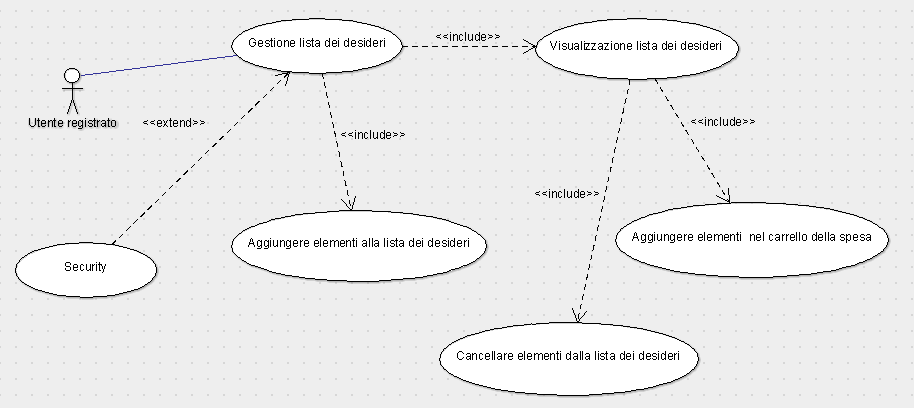
Nella sezione “oders” verranno mostrati tutti gli ordini in stato: aperto, spedito e chiuso e nel caso in cui siano presenti ancora degli item in magazzino si avrà la possibilità di comprarli di nuovo. Il controllo sulla quantità degli item in magazzino è presente in ogni sezione in cui è data all’utente la possibilità di aggiungere item nel carrello.

Nel caso in cui si decida di cancellare o archiviare un ordine prima di procedere ci sarà un pop up per chiedere ulteriore conferma.



### Caso d’uso: Gestione Lista dei desideri (wishlist)

L’utente ha la possibilità di inserire un item presente sulla home o nella sezione “product” nella lista dei desideri anche nel caso in cui non vi sia disponibilità di quell’ oggetto in magazzino. Potrà gestire la sua wishlist tramite l’apposita voce di menù “wishList”.



# Progettazione tecnica

In questo capitolo verranno mostrate le scelte tecniche e le principali informazioni da sapere per installare l’ambiente di sviluppo, far partire l’applicazione ed eventualmente modificarla.

## Quadro generale

La parola microservice è ottenuta come composizione di due parole: *micro* e *service*.

Delle due la più importante è la seconda, *service*, poiché rappresenta il processo storico da cui provengono i microservices ossia il mondo dei servizi; la prima, *micro*, rappresenta l'elemento innovativo che va a rinnovare appunto l'approccio a servizi che proviene dal mondo delle SOA (Service Oriented Architectures) ossia dall'insieme di specifiche internazionali, unitamente ad alcune tecnologie, che ha portato alla definizione dei Web Services. Questi ultimi sono nati essenzialmente come soluzione al problema dell'integrazione tra macrosistemi operanti su piattaforme differenti.

Una cosa da tenere sempre a mente quando si costruisce una architettura microservices è che il risultato finale è una singola applicazione, sia per come funziona sia per come è percepita dagli utenti finali. Ciò significa che, ci deve essere una forte coesione tra i servizi, su come vengono mantenuti e distribuiti e questo al fine di preservare la user experience che l’applicazione si è preposta. Gli aspetti importanti da considerare quando ci si appresta a costruire una architettura Microservices sono:

* Scalabilità: In generale risulta molto più semplice ed economico scalare un microservizio rispetto ad un sistema software monolitico di grandi dimensioni. Il modello a microservizi consente di poter effettuare provisioning delle parti del sistema software in modo dinamico ed intelligente.
* Resilienza: In un’architettura a microservizi, quando una componente non funziona non è automatico che tutto il sistema software smetta di funzionare. In molti casi è possibile isolare il problema ed intervenire mentre il resto del sistema continua a funzionare, cosa non possibile in un’architettura monolitica. Va però sottolineato che l’architettura a microservizi, essendo un insieme di sistemi distribuiti, espone ad una nuova fonte di problemi legati ai disservizi di rete.
* Facilità di deployment: Modificare poche righe di codice su un sistema software monolitico di grandi dimensioni ed effettuarne il deploy è generalmente un’attività non banale, che espone a rischi significativi considerando anche l’impatto che tali modifiche possono avere. Questa paura generalmente porta a raccogliere un certo numero di modifiche prima di avviare un’attività così onerosa e rischiosa. Con l’approccio a microservizi ogni singolo servizio può raggiungere l’ambiente di produzione in modo indipendente, sicché se si verifica un problema esso è facilmente isolato e possono essere intraprese azioni di rollback più velocemente.
* Componibilità: Tra le opportunità più interessanti dell’architettura a microservizi vi è la possibilità di riusare le funzionalità. Infatti è possibile che una stesso servizio venga utilizzato in modi differenti e per scopi diversi. Si pensi ad esempio ad un sistema software che deve poter dialogare non solo col mondo web ma anche con applicazioni mobile, dispositivi wearable, etc.
* Sostituibilità: Quando un sistema software è organizzato a microservizi, il costo di sostituire un servizio con un altro più efficiente e migliore è limitato a circa due settimane di sviluppo, così come banale è il costo di rimuovere un servizio inutile.

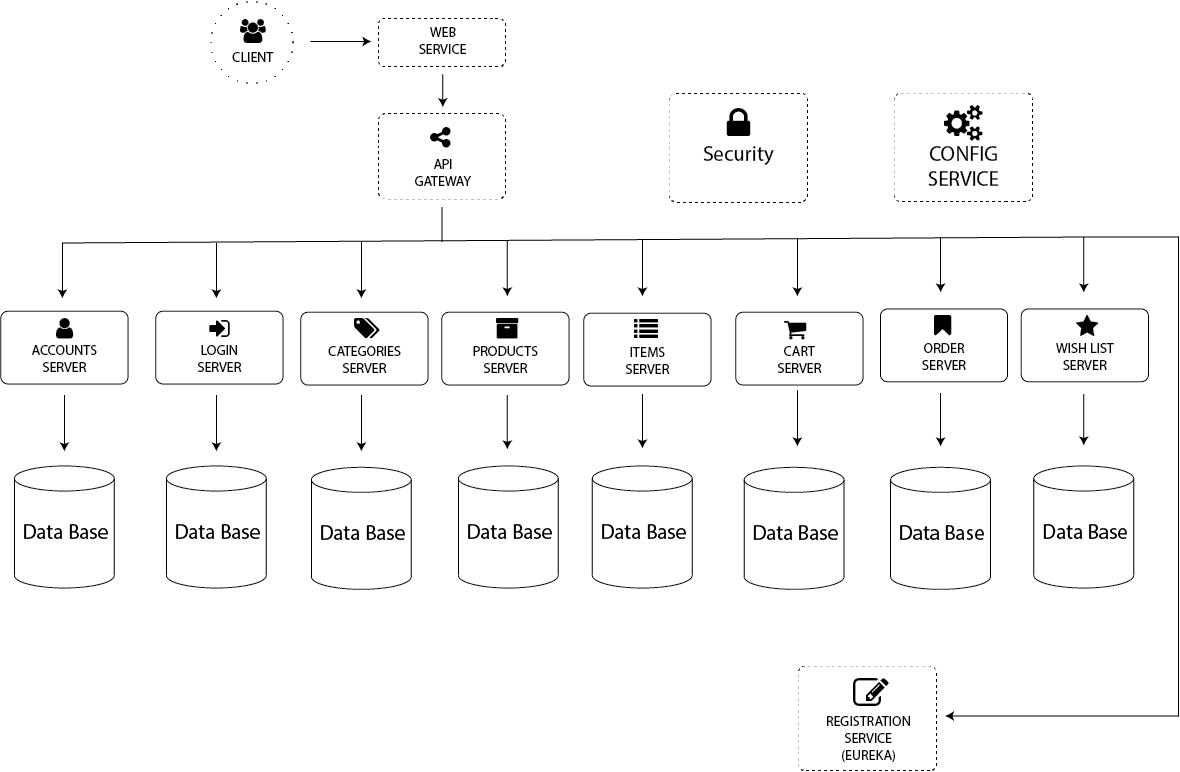
### E-shopper

L’applicazione consta in totale di dodici microservizi e di un modulo .jar per gestire la sicurezza.

Di questi di dodici microservizi: otto sono quelli principali, ognuno indipendente dall’altro e con un proprio database, tre sono quelli necessari per il corretto funzionamento dell’applicativo ed uno è delegato alla parte web.

## SERVIZI INFRASTRUTTURALI

Spring clood offre degli strumenti per migliorare le applicazoni Spring Boot. Tra questi troviamo il servizio di configurazione e il gateway. La figura di seguito mostra come è stata pensata e realizzata infrastrutturalmente l’ applicazione E-shopper.



### Microservice-demo-config

Usato come layer removibile è un servizio per la configurazione centralizzata che permette nel caso di modifiche configurative il riavvio solo di un servizio e non di tutti quelli interessati dalla modifica stessa; potremmo pensarlo come un contenitore di file esterni. Nella cartella *shared* sono presenti tutti i file .yml degli altri microservizi, che al loro interno avranno solo il bootstrap.yml e nessun altro file di tipo .yml, dove vengono definiti il nome dell’ applicazione e l’ url del micricorservice-demo-config. Esempio

spring:

application:

name: account-server

cloud:

config:

uri: http://config:8888

fail-fast: true

Per l’utilizzo lato client è necessario che vi sia la dipendenza con spring-cloud-starter-config all’interno del pom.xml:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

### Microservice-demo-registration

La locazione dei micro-servizi viene rimandata a un nuovo micro-servizio che è responsabile del monitoraggio delle posizioni di tutti gli altri microservices e nient'altro.

Spring Boot mette a disposizione **Netflix’s Eureka** ovvero l'implementazione di un discovery server che permette di registrare i nostri microservizi.

Per creare un discovery server con Spring Boot è sufficiente annotare la classe entry point con *@EnableEurekaServer*:

@EnableEurekaServer

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

Ed aver definito nel pom.xml la dipendenza di spring-cloud-starter-eureka-server:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

Inoltre i nostri microservizi per potersi registrare all’interno del discovery server devono ave annotato nella classe che funge da entry poiny *@EnableDiscoveryClient*  e avere la stessa dipendenza nel pom.xml.

Devono inoltre aver definito nel file .yml le seguenti proprietà:

eureka:

client:

registerWithEureka: true

fetchRegistry: true

serviceUrl:

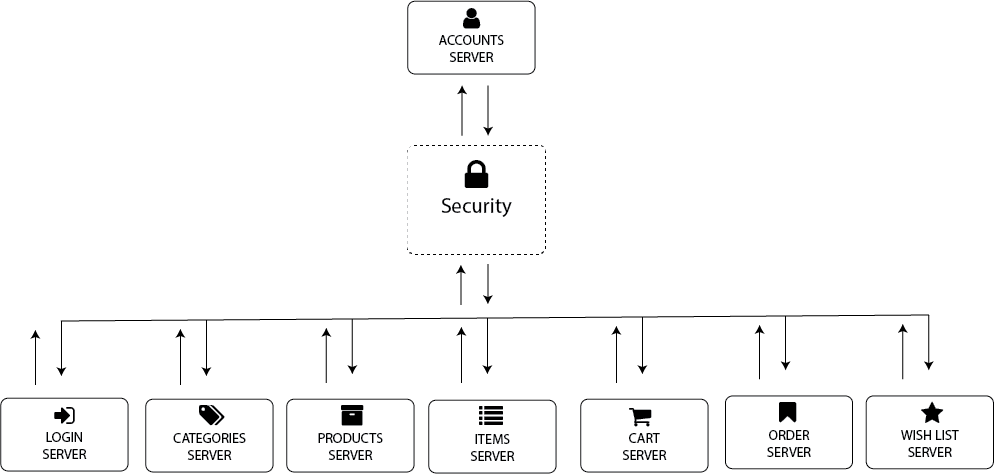
defaultZone: <http://user:password@localhost:1111/eureka/>

Nell’url del defaultZone viene definito anche username e la password per accedere al registry poiché è tutto sotto sicurezza.

Le stesse credenziali (user / password) saranno necessarie anche per accedere da browser a localhost:1111 (ovvero alla discovery server)

### Microservice-demo-security

In realtà questo è l’unico modulo del progetto a non essere un microservizio, ma è una libreria comune utilizzata da tutti i microservizi che a sua volta ne invoca solo uno: microservice-demo-accounts, necessario per effettuare l’autenticazione.



La libreria è stata così implementata per ovviare alla duplicazione di codice e alla logica di business per l’autenticazione.

La sicurezza utilizzata è quella base di spring-boot-starter-security e per utilizzarla nell’applicativo è stato necessario aggiungere in tutti i microservizi:

* la dipendenza nel pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

* il package \*\*.security con all’interno la classe WebSecurityConfig.java in cui vengono definite tra le altre cose tutte le url che non devono essere interessati dall’autenticazione
* All’interno di ogni microservizio (classe entry point annotata con @SpringBootApplication):

nel @ComponentScan il package “it.univaq.ing.security”, questo perché la libreria creata implementa dei bean di Spring

la propagazione dell’autenticazione, tramite la customizzazione del il CustomCredentialProvider utilizzato dall’ HttpClient.

@Bean

@LoadBalanced

public RestTemplate restTemplate(RestTemplateBuilder builder) {

return builder.additionalCustomizers(new AuthCustomizer())

.build();

}

static class AuthCustomizer implements RestTemplateCustomizer {

@Override

public void customize(RestTemplate restTemplate) {

HttpClient client = HttpClientBuilder.create()

.setDefaultCredentialsProvider(new CustomCredentialProvider()).build();

restTemplate.setRequestFactory(new HttpComponentsClientHttpRequestFactory(client));

}

}

Inoltre per una maggiore sicurezza le password sono criptate in SHA-1.

### Microservice-demo-gateway

In una applicazione composta da n servizi potrebbe sembrare naturale che i client chiamino direttamente i servizi che servono per realizzare una determinata funzionalità. Ma questa cosa non sempre risulta essere una soluzione ottimale, in quanto, di norma siamo in presenza di diverse tipologie di client ognuna con le proprie caratteristiche e limitazioni tra cui la necessità di conoscere tutti gli indirizzi degli endpoint o eseguire richieste http differenti e poi unire il risultato su un lato client, ma anche perché la logica diventa troppo legata al client, ad esempio una pagina web di uno store online chiamata da un PC desktop chiamerà sicuramente un numero di servizi superiore rispetto alla stessa pagina chiamata da un app nativa di un dispositivo mobile. Ecco che si è reso necessario introdurre un altro componente con lo scopo di gestire le richieste e per fare da router verso i servizi necessari, appunto **l’API Gateway**. È un singolo punto di ingresso nel sistema, che indirizzerà le richieste al servizio di backend appropriato, invocherà servizi multipli e aggregherà i risultati.

Inoltre eventuali cambi di implementazione di un servizio o l’aggiunta di uno nuovo può avvenire in maniera trasparente per i client. L’API gateway non è considerato di per sé un servizio anche se è buona norma che questo componente risulti il più leggero possibile. Linguaggi di programmazione moderni, come Go o Node.js forniscono dei framework appropriati per costruire API gateway efficienti.

Nel nostro caso abbiamo utilizzato spring-cloud-starter-zuul introducendo nel pom.xml la seguente dipendenza:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zuul</artifactId>

</dependency>

Ed utilizzando l’annotazione @EnableZuulProxy nella classe GatewayApplication.java.

In questo progetto, è stato utilizzato Zuul per conservare i contenuti statici e per girare le richieste ai microservizi appropriati. C'è una semplice regola si routing, basata sui prefissi, per il servizio di accounts per esempio troveremo all’interno del gateway.yml la seguente definizione

routes:

accounts-server:

path: /accounts/\*\*

serviceId: accounts-server

stripPrefix: false

sensitiveHeaders:

Ciò significa che tutte le richieste che iniziano con /accounts saranno instradate verso il microservizio accounts.

Due caratteristiche del API Gateway vanno messe in evidenza: Load balance e Circuit Breaker pattern.

Il load balance è un meccanismo integrato nativamente con Spring Cloud e Server Discovery che si occupa di bilanciare le richieste tra l’elenco dei server disponibili forniti dal microservizio Microservice-demo-registration. Per realizzarlo nella nostra applicazione abbiamo utilizzato **Ribbon.**

Il circuit breaker pattern è un meccanismo per il controllo sulla latenza e l’insuccesso delle dipendenze accessibili dalla rete. L’idea base è quella di mettere a disposizione un gran numero di microservizi così da abbassare notevolmente la probabilità di insuccesso. Per realizzarlo nella nostra applicazione è stato utilizzato **Hystrix**.  
  
Hystrix permette inoltre di aggiungere un metodo di fallback che verrà chiamato per ottenere un valore di default nel caso in cui fallisca il comando principale.

## Ambiente di sviluppo

Per creare l’ambiente di sviluppo sono necessari i seguenti requisiti:

* jdk 1.8
* apache maven 3.5.0
* MySQL 5.6
* IDE STS (spring tool suite), IntelliJ IDEA.

Nel caso in cui una di queste quattro “componenti” sia assente è necessario installarla sul pc.

Dopo aver fatto il clone del progetto da github ed averlo importato all’interno dell IDE è necessario effettuare il maven update project e successivamente il maven clean e il maven install.

### Il database

Un vantaggio di avere un’applicazione distribuita in servizi separati è che ognuno può avere il proprio database.

Nello specifico per ogni microservizio che si interfaccia con il db viene creato uno schema:

CREATE SCHEMA `microservices` DEFAULT CHARACTER SET utf8;

CREATE USER 'microuser'@'localhost' IDENTIFIED BY 'micropassword';

GRANT ALL PRIVILEGES ON \*.microservices TO 'microuser'@'localhost';

il cui puntamento va definito all’interno del progetto, nel file di configurazione *db-config.properties (*…\microservices-application-master\nomeMicroservizio\src\main\resources) con le seguenti proprietà*:*

spring.datasource.url=jdbc:mysql://127.0.0.1:3306/microservices

spring.datasource.username=microuser

spring.datasource.password=micropassword

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

*Per comodità durante la fase di sviluppo è stato utilizzato* ***un solo schema****, motivo per cui i file db-config.properties hanno tutti le stesse proprietà. Lo script per crearlo è presente nel workspace al seguente path: …\microservices-demo.*

Se si decide di utilizzare schema diversi per i microservizi allora bisognerà lanciare questo script n volte avendo cura di modificare: il nome dello schema, username e password dell’utente.

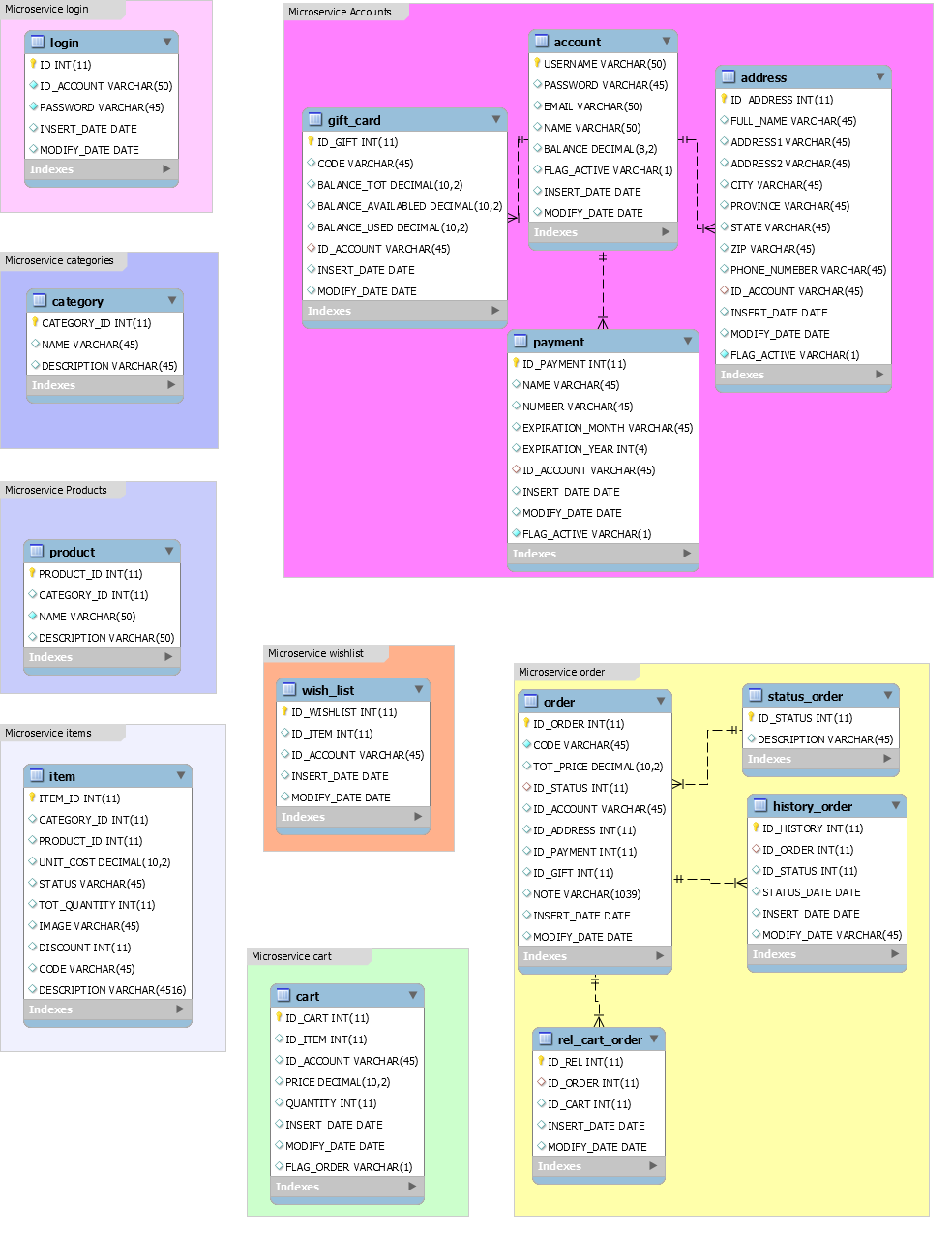
Per ogni microservizio, vengono create le rispettive tabelle del DB. Questa operazione può avvenire in due modi distinti:

* Lanciare dal proprio cliente del db (per esempio MySQL WorkBench 6.3) solo i file create\*\*\*.sql presenti in ognuno dei otto microservizi principali e in seguito avviare i servizi.
* Affidarsi al forward engineering di hibernate. In questo caso non bisogna far nulla prima di avviare i servizi, se non modificare all’interno del db-config.properties la proprietà spring.jpa.hibernate.ddl-auto da *none* a *create.* Lo start dei servizi comporterà anche la creazione delle tabelle. Prima di far vedere l’applicazione da browser sarà comunque necessario lanciare manualmente gli script insert\*\*\*.sql.

Nel workspace i due file (create\*\*\*.sql, insert\*\*\*.sql) sono reperibili all’interno di ogni modulo maven corrispondente ad ognuno degli otto microservizi principali nella directory: …\microservices-application-master\nomeMicroservizio\src\main\resources.

Se le tabelle vengono create tramite script sarà sufficiente lanciare solo il file create\*\*\*.sql, poiché al loro interno sono già presenti gli script per l’inserimento di record nelle tabelle “tipoligiche”.

L’immagine sottostante mostra tutte le tabelle coinvolte nell’applicativo, sono raggruppate per microservizio.



### Come avviare i microservizi

È necessario impostare un ordine di start dei microservizi sia se li avviamo da IDE sia se li avviamo da linea di comando.

* microservice-demo-config (risponde sempre alla porta 8888)
* microservice-demo-registration (risponde sempre alla porta 1111)
* microservice-demo-gateway (risponde sempre alla porta 4000)
* 8 microservizi principali
* microservice-demo-web (risponde sempre alla porta 3333).

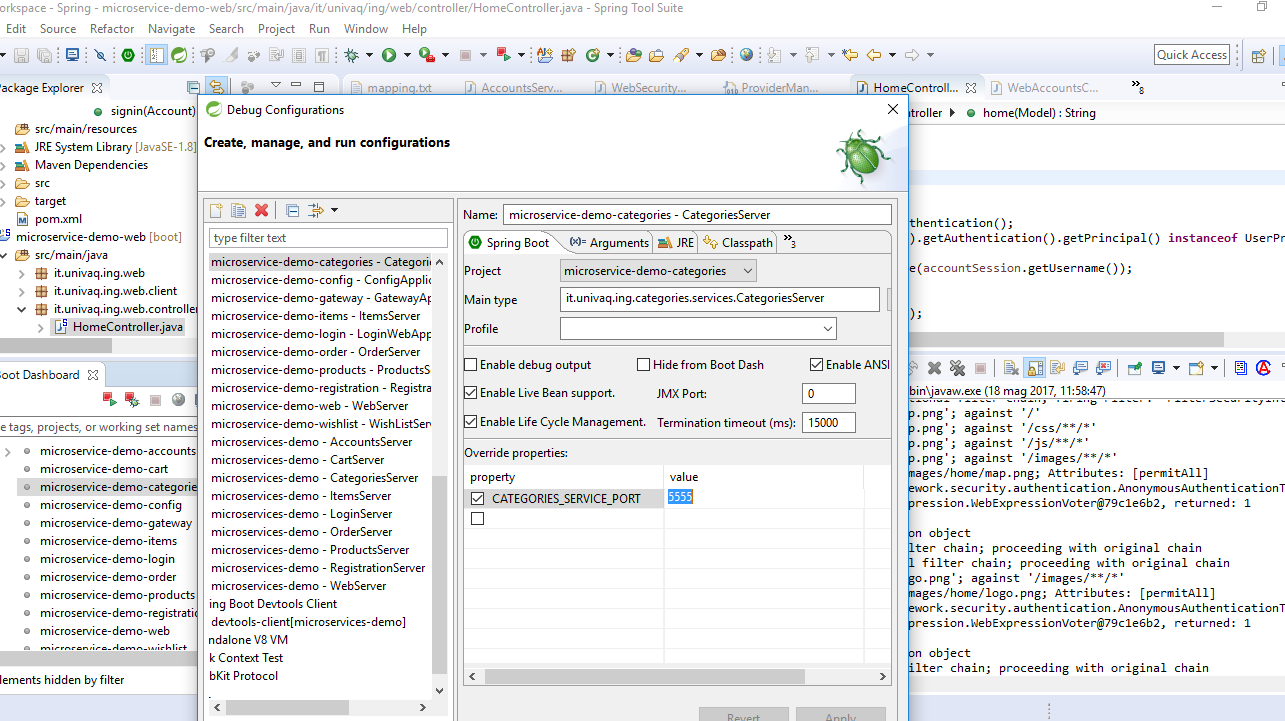
Per gli otto micorservizi principali è stata definita all’interno del corrispondente file .yml nella cartella ‘shared’ del microservice-demo-config una *variabile d’ambiente* per settare la porta a cui rispondono. Esempio per il microservizio account avremo:

server:

context-path: /accounts

port: ${ACCOUNT\_SERVICE\_PORT} # HTTP (Tomcat) port

Se li lanciamo da IDE basta avere configurata la Boot Dashboard ( e configurare il main type ne debug configuration), e configurare per gli 8 servizi principali, la variabile ambiente all’interno della “Debug configuration”.



mentre se li lanciamo da linea di comando bisognerà posizionarsi all’interno della cartella dove risiede ogni microservizio e lanciare indifferentemente i due comandi.

* + java -jar nomemicroservizio.jar (in questo caso occorre posizionarsi dentro target)
  + mvn spring-boot:run

Nel caso degli otto microservizi principali bisogna aggiungere il comando per settare la variabile ambiente:

* + java -DnomeVariabileAmbiente=numeroPorta -jar nomemicroservizio.jar (in questo caso occorre posizionarsi dentro target). Esempio:

java -DACCOUNT\_SERVICE\_PORT=6667 -jar microservice-demo-accounts-0.0.1-SNAPSHOT.jar

* + mvn spring-boot:run -Drun.jvmArguments='- DnomeVariabileAmbiente = numeroPorta’. Esempio:

mvn spring-boot:run -Drun.jvmArguments='-DACCOUNT\_SERVICE\_PORT=6667'

Da notare che nel caso in cui vi siano 2 classi annotate con @SpringBootApplication nello stesso microservizio, sarà necessario definire nel pom con quali delle due dovrà partire. Esempio:

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<version>1.5.3.RELEASE</version>

<configuration>

<mainClass>it.univaq.ing.items.services.ItemsServer</mainClass>

</configuration>

</plugin>

</plugins>

</build>

Definirle anche nel main type nella Debug Configuration della Boot Dashboard dell‘ IDE

Inoltre se si decide di far partire i microservizi solo da linea di comando senza ambiente di sviluppo, prima di procedere con i due comandi sopra descritti e necessario posizionarsi sulla cartella che contiene il parent e lanciare il comando mvn clean install.

Se si desidera avviare più istanze di ogni microservizio principale è sufficiente lanciare i comandi cambiando di volta in volta i numeri di porta e nel caso della Boot Dashboard duplicare la Spring Boot App e definire la variabile ambiente con diverso numero di porta.

## Lo sviluppo

In questo paragrafo vengono evidenziate alcune note sullo sviluppo fatto.

* È stato effettuato un refactoring del progetto in modo tale da avere un modulo maven per ogni microservizio.
* È stata effettuata una pulizia del codice, in modo da diminuire le linee di codice
* È stato reso tutto il progetto dinamico e sono state cancellate tutte le parti statiche
* Sono stati aggiunti oltre agli otto microservizi principali, i microservizi di config e gateway che hanno portato ad un ulteriore refactoring del codice (per esempio sono stati spostati i file .yml nello shared , aggiunti i file bootstrap.yml con conseguente ritest dell’applicativo)
* Sono state gestite tutte le eccezioni. In particolare nei microservizi principali sono state create delle eccezioni custom che estendono RestClientResponseException. Questo per far in modo che nel microservizio web si possano distinguere le varie casistiche con la possibilità di prevedere scenari differenti. Ad esempio nel caso di creazione di un utente, se la mail inserita esiste già, il microservizio accounts rilancerà un’eccezione così definita:

throw new AccountException("existing account",HttpResponseStatus.BAD\_REQUEST.code());

se invece è avvenuto un errore durante la comunicazione con il db si avrà:

throw new AccountException("save account error", HttpResponseStatus.INTERNAL\_SERVER\_ERROR.code());

Lato microservice web si avrà:

ErrorModel err = objectMapper.readValue(((org.springframework.web.client.HttpServerErrorException)e).getResponseBodyAsByteArray(), ErrorModel.class);

if(err.getMessage().equals("existing account")){

model.addAttribute("existenceEmail", Boolean.TRUE );

return "login";

}

model.addAttribute("errorSaveAccount", Boolean.TRUE );

logger.info("ERROR WebAccountsController -->signup: " +e.getMessage());

return "login";

Così l’utente vedrà all’interno dell’ E-shopper due messaggi di errore differente.