Università di Bologna

Progetto di Ingegneria del Software Orientata ai servizi
ACMEsky

Baratin Riccardo
Di Ubaldo Andrea
Vallorani Giacomo

ACMEsky

Progetto di Ingegneria del Software Orientata ai Servizi A.A. 2020/2021

La documentazione web è disponibile al link https://vallasc.github.io/ACMEsky/docs/

Descrizione del dominio e del problema

ACMESky offre un servizio di che permette ai clienti di specificare, attraverso un portale web, il proprio interesse a trasferimenti aerei di andata e ritorno che si tengano in un periodo definito e ad un costo inferiore ad un certo limite impostato.

ACMESky quotidianamente interroga le compagnie aeree per ottenere le quotazioni dei voli di interesse per i propri clienti.

ACMESky riceve anche offerte last-minute dalle compagnie che le inviano al momento dell'attivazione senza cadenza prefissata.

Quando ACMESky trova un volo compatibile con una richieste di un cliente prepara un'offerta.

L'offerta viene inviata al cliente tramite l'APP di messaggistica Prontogram. Il cliente, se interessato, ha quindi 24 ore di tempo per connettersi al portale web di ACMESky per confermare l'offerta, specificandone il codice ricevuto via Prontogram.

In fase di conferma il cliente deve anche procedere al pagamento, per la gestione del quale ACMESky si appoggia ad un fornitore di servizi bancari: ACMESky reindirizza il cliente verso il sito del fornitore e quindi attende dal fornitore il messaggio che conferma l'avvenuto pagamento.

Nel caso in cui il costo del volo risulti essere superiore ai 1000 euro ACMESky offre al cliente un servizio gratuito di trasferimento da/verso l'aeroporto se questo si trova entro i 30 chilometri dal suo domicilio.

In questo caso ACMESky fa uso di diverse compagnie di noleggio con autista con cui ha degli accordi commerciali. La compagnia scelta è quelle che risulta avere una sede più vicina al domicilio del cliente. A tale compagnia ACMESky invia una richiesta per prenotare un trasferimento che parta due ore prima dell'orario previsto per il decollo del volo.

Vincoli al problema

- Le offerte generate provengono dalla stessa compagnia aerea
- Non viene gestito lo scambio di denaro dalla banca alla compagnia aerea
- Se uno dei servizi per la gestione del calcolo premium non risponde i voli vengono acquistati senza servizi accesori

- Ogni attore deve essere autenticato per poter interagire con ACMEsky

La soa è composta dai seguenti servizi:

TODO TODO

Struttura della relazione

- Coreografie
- BPMN
- Coreografie BPMN
- UML
- Struttura SOA
- Esecuzione
- Servizi
    - ACMEsky
        - Service
        - Database
        - Web
    - AirlineService
    - BankService
    - Prontogram
    - RentService
    - GeographicalDistance

TODO Spostare in STRUTTURA

Service	Port
ACMEsky	-
web	80
api	8080
Postgres	5050
BankService	8070
AirlineNationalService	8060

AirlineService	8061
ProntogramService	8050
ProntogramFrontEnd	8051
GeocraphicalService	8040
RentService1	8030
RentService2	8032

Coreografie

In questa sezione vengono mostrate le coreografie.

Nomenclatura

Nome	Sigla	Commento
ACME	ACME	
Airline company	AIR_k	Indica la k-esima compagnia aerea
Rent company	$RENT_t$	Indica la t-esima compagnia di noleggio
Prontogram	PTG	
Bank	BANK	
Geodistance service	GEO	
USER	$USER_x$	Indica l'x-esimo utente

Coreografia complessiva del sistema

```
// Query dei voli (ripetuta per tutte le compagnie aeree)
// Viene ripetuta per ogni compagnia aerea collegata ad ACMEsky
// queryFlights: Richesta di voli d'interesse per l'utente
// responseFlights: Voli disponibili dell'Airline company
( queryFlights: ACME -> AIRₖ ; responseFlights: AIRₖ -> ACME )*
|

// Ricezione offerte last minute (ripetuta per tutte le compagnie aeree)
// Viene ripetuta per ogni compagnia aerea collegata ad ACMEsky
// sendLastMinute: invia le offerte last minute
// repsponseLastMinute: risposta successo o fallimento
( sendLastMinute: AIRₖ -> ACME ; repsponseLastMinute: ACME -> AIRₖ )*
|

// Registrazione interesse dell'utente (ripetuta per tutti gli  utenti)
// requestInterest: messaggio di richiesta con A/R
// responseInterest: risposta successo o fallimento
( requestInterest: USERₓ -> ACME ; responseInterest: ACME -> USERₓ )*
|
```

```
// Notifica dell'offerta all'utente
// offerToken: mesaagio di offerta A/R
// notifyUser: messaggio di notifica di Prontogram
// notifyResponse: risposta da parte dell'utene dell'avvenuta ricezione
// messageSended: risposta da parte di prontogram dell'avvenuto invio
( offerToken: ACME -> PTG ; notifyUser: PTG -> USER_x ;
  notifyResponse: USER_x -> PTG ; messageSended: PTG -> ACME )*
|

// Richiesta ticket
// getInvoice: mesaagio di richiesta ricevuta dell'offerta pagata
// invoice: messaggio con la ricevuta del viaggio
( getInvoice: USER_x -> ACME ; invoice: ACME -> USER_x )*
|

// Conferma dell'offerta e pagamento
// confirmOffer: messaggio di conferma offerta e pagamento
(
  confirmOffer: USER_x -> ACME ;
  (
    // ACMEsky conferma che l'offerta è disponibile
    // responseOfferOk: messaggio di conferma offerta
    // requestPaymentLink: richiesta di pagamento da parte dell'utente
    (
      responseOfferOk: ACME -> USER_x ;
      requestPaymentLink: USER_x -> ACME ;
      (
        // Tickets ok
        // bookTickets: prenota i biglietti
        // responseTickets: biglietti prenotati
        // requestBankLink: richiesta creazione link di pagamento
        // responselink: link di pagamento generato dalla banca
        // paymentLink: link di pagamento generato dalla banca
        // payment: pagamento attraverso il link generato
        (
          bookTickets: ACME -> AIR_k ;
          responseTickets: AIR_k -> ACME ;
          requestBankLink: ACME -> BANK ;
          responselink: BANK -> ACME ;
          paymentLink: ACME -> USER_x ;
          payment: USER_x -> BANK ;
          (
            // Pagamento avvenuto con successo
            // successPaymentBank: esito pagamento
            (
              successPaymentBank: BANK -> ACME ;
              // Controllo Premium service
              (
                // Richiesta a Geodistance se costo > 1000€
                1
                +
                // requestDistance: richiesta calcolo della distanza
                // responseDistance: distanza calcolata
                (
                  requestDistance: ACME -> GEO ;
```

```
                responseDistance: GEO -> ACME ;
                ( // Richiesta a Rent service se distanza <30Km
                  1
                  +
                  (
                    (
                      // requestDistanceRent: richiesta distanza noleggio
                      // responseDistanceRent: risposta con distanza
                      requestDistanceRent: ACME -> GEO ;
                      responseDistanceRent: GEO -> ACME
                    )* ;
                    // requestRentDeparture: richiesta noleggio andata
                    // responseRentDeparture: risposta nolleggio andata
                    // requestRentReturn: richiesta noleggio ritorno
                    // responseRentReturn: risposta nolleggio ritorno
                    requestRentDeparture: ACME -> RENT_t ;
                    responseRentDeparture: RENT_t-> ACME ;
                    requestRentReturn: ACME -> RENT_t ;
                    responseRentReturn: RENT_t-> ACME
                  )
                )
              )
            )
          )
          +
          (
            // Errore nel pagamento
            // unbookTickets: cancella la prenotazione dei biglietti
            // emitCoupon: pagamento fallito
            unbookTickets: ACME -> AIR_k ;
            emitCoupon: ACME -> BANK
          )
        )
      )
      // Errore nella prenotazione dei biglietti
      // errorTickets: errore volo non disponibile
      +
      errorTickets: ACME -> USER_x
    )
  )
  +
  // ACMEsky controlla l'offerta e non è più disponibile
  // responseOfferError: errore offerta
  responseOfferError: ACME -> USER_x
)
)*
```

Verifica condizioni connectedness delle coreografie

Analizzando la coreografia si nota che essa fa parte del caso asincrono. Per stabilire la connectedness, e per una migliore lettura, la coreografia è stata divisa in 6 blocchi:

1. **Query dei voli**
2. **Ricezione offerte last-minute**
3. **Registrazione interesse dell'utente**
4. **Notifica dell'offerta all'utente**
5. **Richiesta della tickets dell'offerta**
6. **Conferma dell'offerta e pagamento**

Essendo queste sotto-coreografie eseguite in parallelo non ci sono condizioni da rispettare, pertanto, si è passati a valutare la correttezza di ogni singolo blocco.

Query dei voli

```
( queryFlights: ACME -> AIR_k ; responseFlights: AIR_k -> ACME )*
```

E' connessa per la sequenza in quanto il ricevente in ***queryFlights*** è il mittente di ***responseFlights***. E' connessa anche per l'iterazione in quanto il ricevente in ***responseFlights*** è il mittente di ***queryFlights***.

Ricezione offerte last-minute

```
( sendLastMinute: AIR_k -> ACME ; repsponseLastMinute: ACME -> AIR_k )*
```

E' connessa in quanto il ricevente in ***sendLastMinute*** è il mittente di ***repsponseLastMinute***. E' connessa anche per l'iterazione in quanto il ricevente in ***repsponseLastMinute*** è il mittente di ***sendLastMinute***.

Registrazione interesse dell'utente

```
( requestInterest: USER_x -> ACME ; responseInterest: ACME -> USER_x )*
```

E' connessa in quanto il ricevente in ***requestInterest*** è il mittente di ***responseInterest***. E' connessa anche per l'iterazione in quanto il ricevente in ***responseInterest*** è il mittente di ***requestInterest***.

Notifica dell'offerta all'utente

```
( offerToken: ACME -> PTG ; notifyUser: PTG -> USER_x ; notifyResponse: USER_x -> PTG ;
messageSended: PTG -> ACME )*
```

E' connessa in quanto il ricevente in ***offerToken*** è il mittente di ***notifyUser***, il ricevente in ***notifyUser*** è il mittente di ***notifyResponse***, il ricevente in ***notifyResponse*** è il mittente di ***messageSended***. E' connessa

anche per l'iterazione in quanto il ricevente in ***messageSended*** è il mittente di ***offerToken***.

Richiesta ricevuta dell'offerta

```
( getInvoice: USER_x -> ACME ; invoice: ACME -> USER_x )*
```

E' connessa in quanto il ricevente in ***getInvoice*** è il mittente di ***invoice***. E' connessa anche per l'iterazione in quanto il ricevente in ***invoice*** è il mittente di ***getInvoice***.

Conferma dell'offerta e pagamento

```
1. ( confirmOffer: USER_x -> ACME ;
```

E' connessa in quanto il ricevente di ***confirmOffer*** è il mittente di **(3)** e di **(27)**.

```
2.   (
3.     ( responseOfferOk: ACME -> USER_x ; requestPaymentLink: USER_x -> ACME ;
```

E' connessa per la sequenza in quanto il ricevente di ***requestPaymentLink*** è il mittente di **(6)**.

```
4.       (
5.         (
6.           bookTickets: ACME -> AIR_k ; responseTickets: AIR_k -> ACME ;
7.           requestBankLink: ACME -> BANK ; responselink: BANK -> ACME ;
8.           paymentLink: ACME -> USER_x ;
9.           payment: USER_x -> BANK ;
```

E' connessa per la sequenza in quanto il ricevente di ***bookTickets*** è il mittente di ***responseTickets***, il ricevente di ***requestBankLink*** è il mittente di ***responselink***, il ricevente di ***responselink*** è il mittente di ***paymentLink*** e il ricevente di ***paymentLink*** è il mittente di ***payment***.

Inoltre, è connessa per la choice perché il destinatario di ***payment*** è il mittente di **(11)** e di **(21)**

```
10.          (
11.            ( successPaymentBank: BANK -> ACME ;
```

E' connessa per la choice perché il destinatario di ***successPaymentBank*** è il mittente di **(13)**

```
12.             ( 1 +
13.               ( requestDistance: ACME -> GEO ; responseDistance: GEO -> ACME ;
```

E' connessa per la sequenza perché il destinatario di ***requestDistance*** è il mittente di ***responseDistance***. E' connessa per la choice perché il destinatario di ***responseDistance*** è il mittente di **(14)**

```
14.                 ( 1 +  ( ( requestDistanceRent: ACME -> GEO ; responseDistanceRent: GEO ->
ACME )* ;
```

E' connessa per la sequenza perché il destinatario di ***requestDistanceRent*** è il mittente di ***responseDistanceRent***. E' connessa per l'iterazione perché destinatario di ***responseDistanceRent*** è il mittente di ***requestDistanceRent***, quindi la coreografia può essere iterata.

```
15.                     requestRentDeparture: ACME -> RENT_t ; responseRentDeparture: RENT_t-> ACME
;
16.                     requestRentReturn: ACME -> RENT_t ; responseRentReturn: RENT_t-> ACME
17                    )
18.                 )
19.               )
20.             )
```

E' connessa per la sequenza perché il destinatario di ***responseDistanceRent*** è il mittente di ***requestRentDeparture***, il destinatario di ***requestRentDeparture*** è il mittente di ***responseRentDeparture***, il destinatario di ***responseRentDeparture*** è il mittente di ***requestRentReturn*** e il destinatario di requestRentReturn è il mittente di ***responseRentReturn***.

```
21.             ) + ( errorPaymentBank: BANK -> ACME ;
22.                   unbookTickets: ACME -> AIR_k ; unbookTicketsResponse: AIR_k -> ACME ;
23.                   emitCoupon: ACME -> BANK ; emitCouponResponse: BANK -> ACME
                )
```

E' connessa per la sequenza perché il destinatario di ***errorPaymentBank*** è il mittente di ***unbookTickets***, il destinatario di ***unbookTickets*** è il mittente di ***unbookTicketsResponse***, il destinatario di ***emitCoupon*** è il mittente di ***emitCouponResponse***.

```
24.           )
25.         ) + errorTickets: ACME -> USER_x
```

E' connessa per la choice perchè i mittenti di **(6)** e di **(25)** sono gli stessi.

```
26.       )
27.     ) + responseOfferError: ACME -> USER_x
```

```
28.   )
```

E' connessa per la choice i sender di **(3)** e di **(27)** sono gli stessi.

```
29. )*
```

La coreografia è connessa per l'iterazione in quanto **(25)** e **(27)** terminano con il ricevente **USER** che è il mittente di **(1)**, mentre **(24)** termina con **ACME** che è connessa con **(1)** secondo il pattern Receiver.

Proiezioni

ACMEsky

```
proj(QueryDeiVoli, ACME) =
    ____________
  ( queryFlights@AIR_k ; responseFlights@AIR_k )*
```

```
proj(RicezioneOfferteLastMinute, ACME) =
                          ___________________
  ( sendLastMinute@AIR_k ; repsponseLastMinute@AIR_k )*
```

```
proj(RegistrazioneInteresse, ACME) =
                            _________________
  ( requestInterest@USER_x ; responseInterest@USER_x )*
```

```
proj(NotificaOfferta, ACME) =
    __________
  ( offerToken@PTG ; 1 ; 1 ; messageSended@PTG )*
```

```
proj(RichiestaRicevuta, ACME) =
                       ________
  ( getInvoice@USER_x ; invoice@USER_x )*
```

```
proj(AcquistoOfferta, ACME) =
  ( confirmOffer@USER_x ;
```

```
    (
      (responseOfferOk@USER_x ; requestPaymentLink@USER_x ;
        (
          ( ____________
            bookTickets@AIR_k ; responseTickets@AIR_k ;
            _______________
            requestBankLink@BANK ; responselink@BANK ;
            ____________
            paymentLink@USER_x ; 1 ;
            (
              (
                successPaymentBank@BANK ;
                        ________________
                ( 1 + ( requestDistance@GEO ; responseDistance@GEO ;
                              ____________________
                  ( 1 + ( ( requestDistanceRent@GEO ; responseDistanceRent@GEO )* ;
                    ________________________
                    requestRentDeparture@RENT_t ; responseRentDeparture@RENT_t ;
                    ___________________
                    requestRentReturn@RENT_t ; responseRentReturn@RENT_t
                  ) )
                ) ) _______________          ____________
              ) + ( unbookTickets@AIR_k ; emitCoupon@BANK )
            )
              _____________
          ) + errorTickets@USER_x
        )
          ____________________
      ) + responseOfferError@USER_x
    )
  )*
```

Utente

```
proj(QueryDeiVoli, USER_x) =
  ( 1 ; 1 )*
```

```
proj(RicezioneOfferteLastMinute, USER_x) =
  ( 1 ; 1 )*
```

```
proj(RegistrazioneInteresse, USER_x) =
    _________________
  ( requestInterest@ACME ; responseInterest@ACME )*
```

```
proj(NotificaOfferta, USER_x) =
                    _________________
  ( 1 ; notifyUser@PTG ; notifyResponse@PTG ; 1 )*
```

```
proj(RichiestaRicevuta, USER_x) =
    __________
  ( getInvoice@ACME ; invoice@ACME )*
```

```
proj(AcquistoOfferta, USER_x) =
    ____________
  ( confirmOffer@ACME ;
    (                          __________________
      ( responseOfferOk@ACME ; requestPaymentLink@ACME ;
        (
          ( 1 ; 1 ; 1 ; 1 ; paymentLink@ACME ; payment@BANK ;
            (
              (
                1 ;
                ( 1 + ( 1 ; 1 ;
                  ( 1 + (( 1 ; 1)* ; 1 ; 1 ; 1 ; 1 ))
                ))
              ) + ( 1 ; 1 )
            )
          ) + errorTickets@ACME
        )
      ) + responseOfferError@ACME
    )
  )*
```

Airline

```
proj(QueryDeiVoli, AIR_k) =
                      _______________
  ( queryFlights@ACME ; responseFlights@ACME )*
```

```
proj(RicezioneOfferteLastMinute, AIR_k) =
    ________________
  ( sendLastMinute@ACME ; repsponseLastMinute@ACME )*
```

```
proj(RegistrazioneInteresse, AIR_k) =
  ( 1 ; 1 )*
```

```
proj(NotificaOfferta, AIR_k) =
  ( 1 ; 1 ; 1 ; 1 )*
```

```
proj(RichiestaRicevuta, AIRₖ) =
  ( 1 ; 1 )*
```

```
proj(AcquistoOfferta, AIRₖ) =
  ( 1 ;
    (
      (1 ; 1 ;
        (
          (                    _______________
            bookTickets@ACME ; responseTickets@ACME ;
            1 ; 1 ; 1 ; 1 ;
            (
              ( 1 ;
                ( 1 + ( 1 ; 1 ;
                  ( 1 + (( 1 ; 1)* ; 1 ; 1 ; 1 ; 1 ))
                ))
              ) + ( unbookTickets@ACME ; 1 )
            )
          ) + 1
        )
      ) + 1
    )
  )*
```

Prontogram

```
proj(QueryDeiVoli, PTG) =
  ( 1 ; 1 )*
```

```
proj(RicezioneOfferteLastMinute, PTG) =
  ( 1 ; 1 )*
```

```
proj(RegistrazioneInteresse, PTG) =
  ( 1 ; 1 )*
```

```
proj(NotificaOfferta, PTG) =
                       __________
  ( offerToken@ACME ; notifyUser@USERₓ ;
                                 _______________
    notifyResponse@USERₓ ; messageSended@ACME )*
```

```
proj(RichiestaRicevuta, PTG) =
  ( 1 ; 1 )*
```

```
proj(AcquistoOfferta, PTG) =
  ( 1 ;
    (
      ( 1 ; 1 ;
        (
          ( 1 ; 1 ; 1 ; 1 ; 1 ; 1 ;
            (
              ( 1 ;
                ( 1 + ( 1 ; 1 ;
                  ( 1 + (( 1 ; 1)* ; 1 ; 1 ; 1 ; 1 ))
                ))
              ) + ( 1 ; 1 )
            )
          ) + 1
        )
      ) + 1
    )
  )*
```

Bank

```
proj(QueryDeiVoli, BANK) =
  ( 1 ; 1 )*
```

```
proj(RicezioneOfferteLastMinute, BANK) =
  ( 1 ; 1 )*
```

```
proj(RegistrazioneInteresse, BANK) =
  ( 1 ; 1 )*
```

```
proj(NotificaOfferta, BANK) =
  ( 1 ; 1 ; 1 ; 1 )*
```

```
proj(RichiestaRicevuta, BANK) =
  ( 1 ; 1 )*
```

```
proj(AcquistoOfferta, BANK) =
  ( 1 ;
    (
      ( 1 ; 1 ;
        (
          ( 1 ; 1 ;
                                  ____________
            requestBankLink@ACME ; responselink@ACME ;
            1 ; payment@USER_x ;
            (
              ( ____________________
                successPaymentBank@ACME ;
                        _________________
                ( 1 + ( 1 ; 1 ;
                  ( 1 + (( 1 ; 1)* ; 1 ; 1 ; 1 ; 1 ))
                ))
              ) + ( 1 ; emitCoupon@ACME )
            )
          ) + 1
        )
      ) + 1
    )
  )*
```

Geodistance service

```
proj(QueryDeiVoli, GEO) =
  ( 1 ; 1 )*
```

```
proj(RicezioneOfferteLastMinute, GEO) =
  ( 1 ; 1 )*
```

```
proj(RegistrazioneInteresse, GEO) =
  ( 1 ; 1 )*
```

```
proj(NotificaOfferta, GEO) =
  ( 1 ; 1 ; 1 ; 1 )*
```

```
proj(RichiestaRicevuta, GEO) =
  ( 1 ; 1 )*
```

```
proj(AcquistoOfferta, GEO) =
  ( 1 ;
    (
      ( 1 ; 1 ;
        (
          ( 1 ; 1 ; 1 ; 1 ; 1 ; 1 ;
            (
              ( 1 ;
                                                  _________________
                ( 1 + ( requestDistance@ACME ; responseDistance@ACME ;
                                                                    ____________________
                  ( 1 + (( requestDistanceRent@ACME ; responseDistanceRent@ACME )* ;
                    1 ; 1 ; 1 ; 1 ))
                ))
              ) + ( 1 ; 1 )
            )
          ) + 1
        )
      ) + 1
    )
  )*
```

Rent company

```
proj(QueryDeiVoli, RENT_t) =
  ( 1 ; 1 )*
```

```
proj(RicezioneOfferteLastMinute, RENT_t) =
  ( 1 ; 1 )*
```

```
proj(RegistrazioneInteresse, RENT_t) =
  ( 1 ; 1 )*
```

```
proj(NotificaOfferta, RENT_t) =
  ( 1 ; 1 ; 1 ; 1 )*
```

```
proj(RichiestaRicevuta, RENT_t) =
  ( 1 ; 1 )*
```

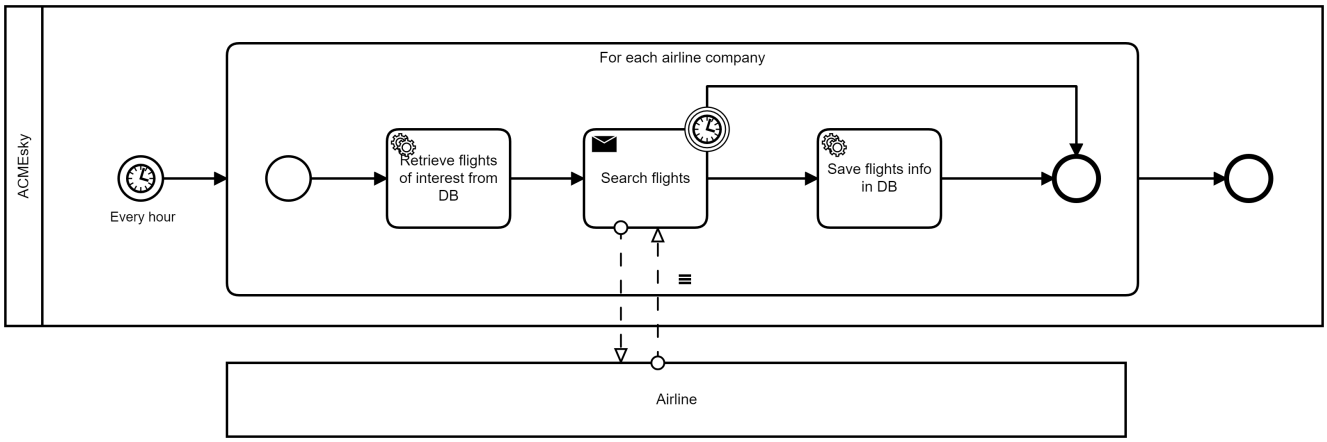
```
proj(AcquistoOfferta, RENT_t) =
  ( 1 ;
    (
```

```
      ( 1 ; 1 ;
        (
          ( 1 ; 1 ; 1 ; 1 ; 1 ; 1 ;
            (
              ( 1 ;
                ( 1 + ( 1 ; 1 ;
                  ( 1 + (( 1 ; 1)* ;
                                                  ____________________
                    requestRentDeparture@ACME ; responseRentDeparture@ACME ;
                                                  ___________________
                    requestRentReturn@ACME ; responseRentReturn@ACME ))
                ))
              ) + ( 1 ; 1 )
            )
          ) + 1
        )
      ) + 1
    )
  )*
```

Diagramma BPMN

In questa sezione della documentazione viene descritto il diagramma BPMN della *Service Oriented Architecture* (SOA), nel quale viene mostrato come i servizi ed **ACMEsky** interagiscono tra loro al fine di realizzare le funzionalità richieste. Per una migliore specificità e gestione della documentazione il diagramma verrà diviso in parti relative alle varie azioni degli attori, come ad esempio: la registrazione dell'interesse utente, la ricerca dei voli, salvataggio di quelle last-minute, gestione delle offerte e pagamento, ecc.

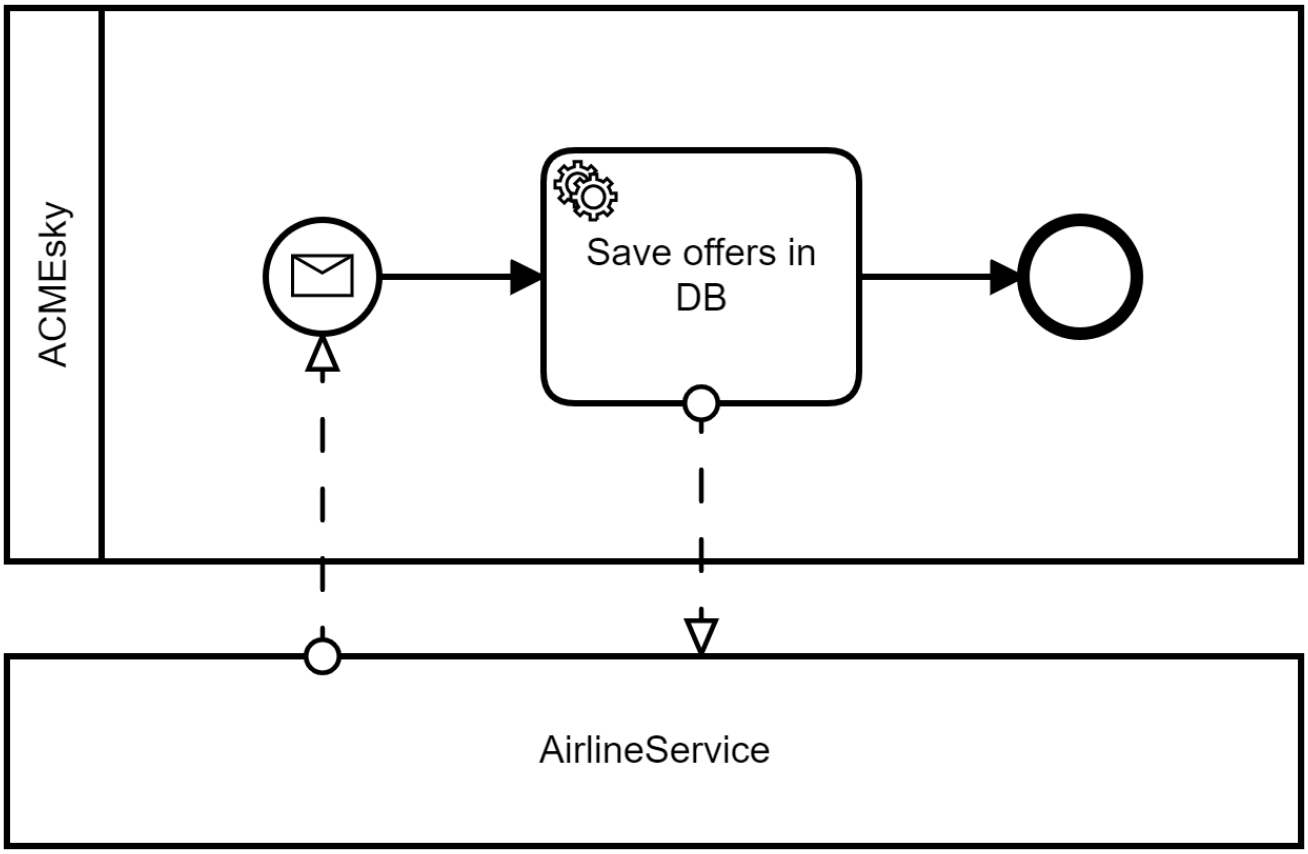
Ricerca voli



La ricerca delle dei voli alle viene ripetuta con un certo intervallo di tempo, per evitare sovraccaricare i sistemi. Per questo i voli delle offerte di interesse degli utenti vengono salvate sul DB per "*bufferizzarle*", in modo da non perderle ed utilizzarle per cercare i voli delle compagnie aeree. L'intervallo di tempo è variabile da 1 ora a pochi minuti poichè si cerca un compromesso tra un sistema efficiente e un sistema che non faccia aspettare troppo l'utente.

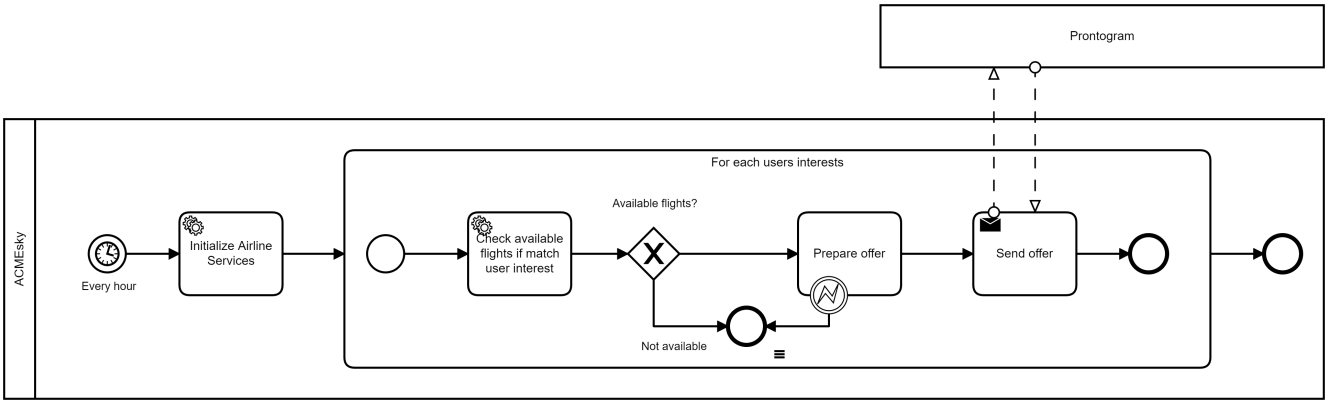
Quindi, ad un certo intervallo e per ciascun **Airline Service**, **ACMEsky** recupera i voli di interesse degli utenti dal suo Database ed effettua una ricerca mirata dei voli compatibili con essi attraverso una chiamata ad una risorsa dei servizi di volo. I voli presenti nella risposta vengono salvati all'interno del database (tabella *available_flights*). Se il timer della richiesta scade, per cause dovute ad **Airline Service**, il sottoprocesso termina e si passa ad un'altra compagnia.

Registrazione delle offerte last-minute



In questa parte si descrive il processo di ricezione e salvataggio dei voli last-minute. I servizi di **Airline Service** mandano voli last-minute appena creati ad **ACMEsky**, la quale salva ciascuno di essi nel database, nello specifico nella tabella *available_flights*.

Match voli con interesse utente

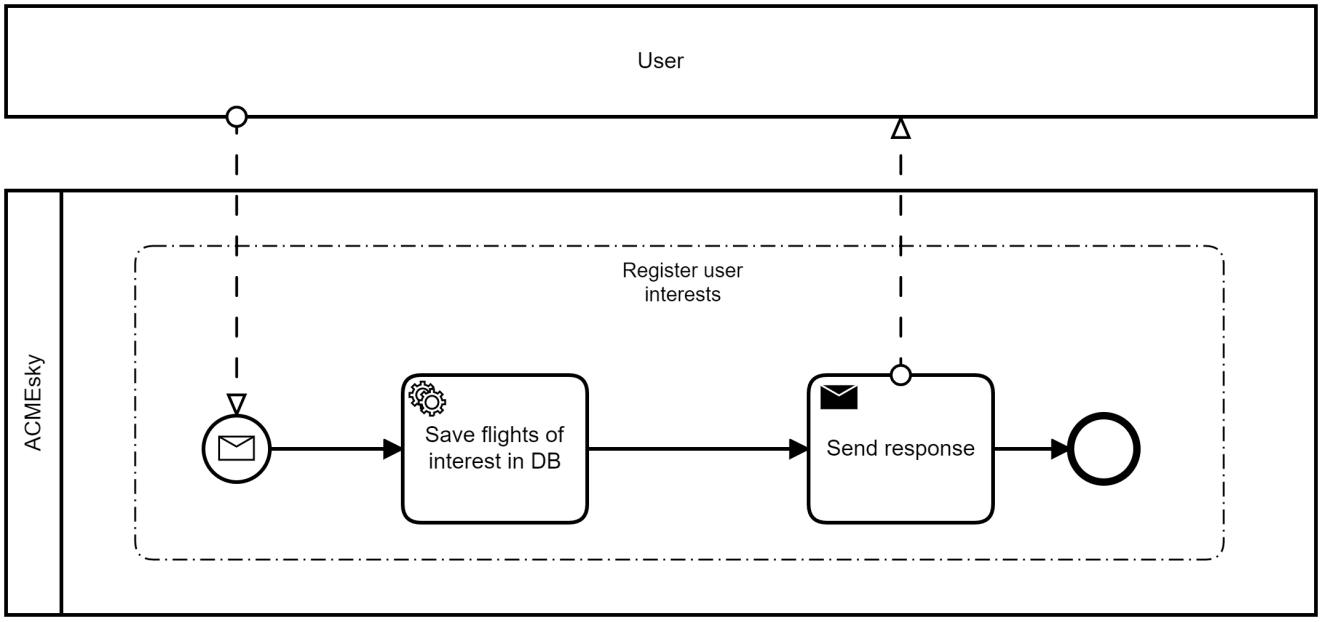


La generazione delle offerte di volo viene fatta ad un certo intervallo di tempo, ciò consente di non sovraccaricare il sistema e di evitare problemi di concorrenza con altri processi che generano le offerte. Per questo motivo, i voli delle offerte di interesse degli utenti vengono salvati sul database finchè non vengono

utilizzati dal processo. L'intervallo di tempo è variabile da un'ora a pochi minuti poichè si cerca un compromesso tra un sistema efficiente e un sistema che non faccia aspettare l'utente.

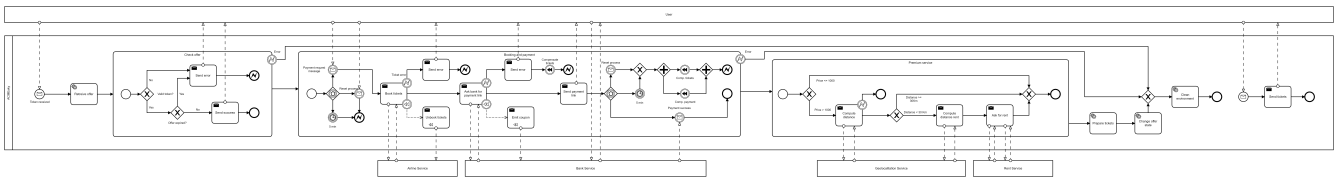
Ogni ora, per ciascun offerta di interesse, **ACMEsky** cerca tra i voli disponibili presenti nel database (tabella *available_flights*), se c'è una corrispondenza con l'interesse dell'utente allora prepara l'offerta, la salva nel database e la invia all'utente attraverso **Prontogram**. In caso negativo semplicemente il flusso termina passando all'interesse successivo.

Registrazione dell'interesse dell'utente



Il seguente diagramma descrive il processo di raccolta e registrazione degli interessi degli utenti. L'utente descrive i suoi voli di interesse specificando: città/aereoporto di partenza, città/aereoporto di arrivo, data di partenza, data ritorno e quota massima di spesa. **ACMEsky** salva i voli di interesse nel suo database, in particolare, nella tabella *flights_interest* e in *users_interests*, che contiene l'interesse per uno specifico viaggio. Infine, **ACMEsky** invia la conferma di avvenuta creazione.

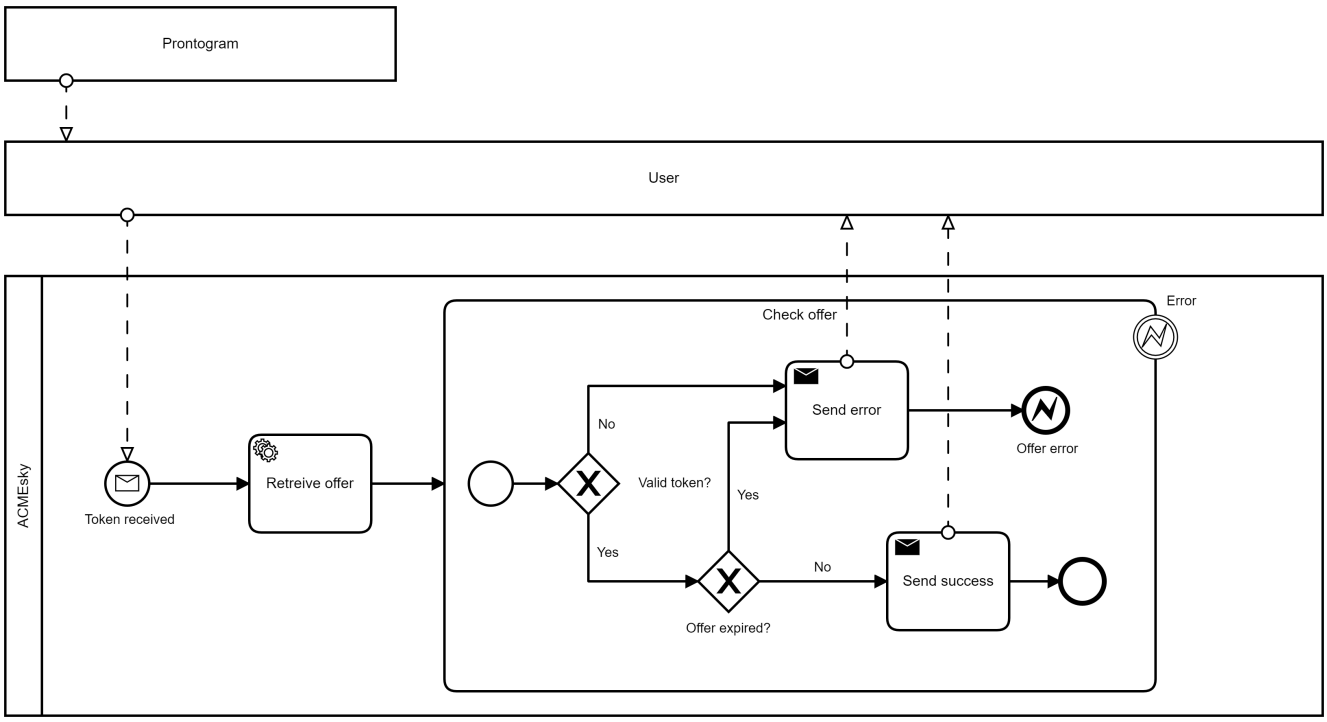
Conferma di acquisto, applicazione servizi premium e preparazione biglietti



In questa parte del diagramma viene illustrata la conferma di acquisto dell'offerta da parte dell'utente, l'acquisto dell'offerta e l'applicazione dei servizi premium se l'offerta rispetta le caratteristiche necessarie. Infine, viene preparato il biglietto che poi l'utente potrà scaricare.

Per una magiore comprensione il diagramma è stato diviso in blocchi più piccoli.

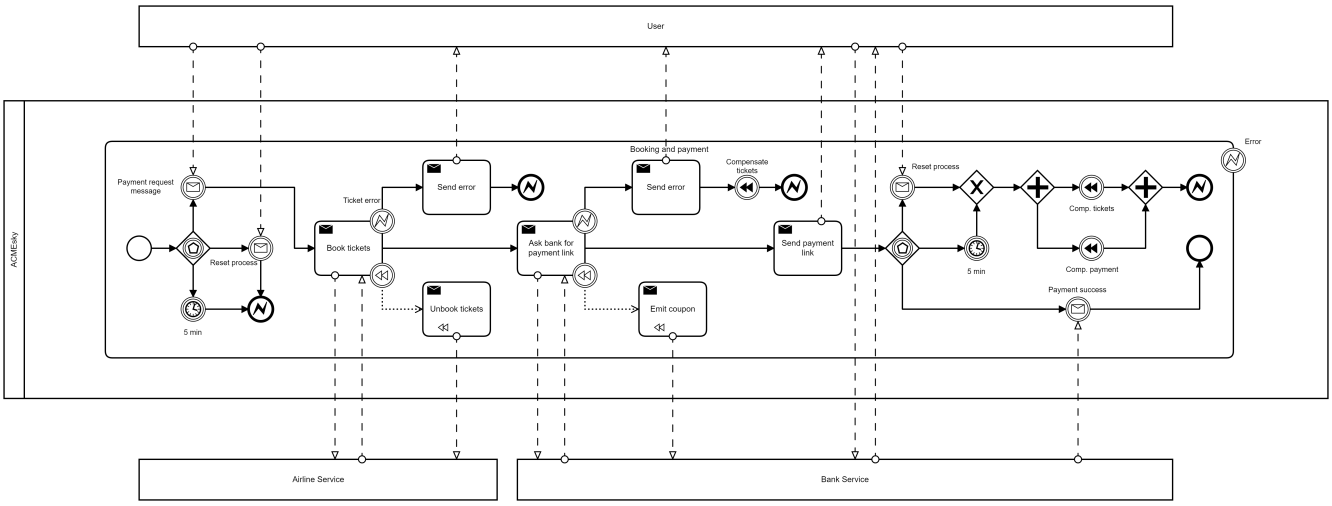
Conferma dell'offerta da parte dell'utente



L'app di **Prontogram** notifica l'utente del fatto che c'è un'offerta disponibile.

L'utente riceve l'offerta e può decidere se confermarla o meno attraverso l'invio di un token legato all'offerta stessa. **ACMEsky** recupera l'offerta corrispondente al token e si occupa di verificarne la validità, ovvero, di controllare che il tempo di accettazione dell'offerta non sia terminato. In caso positivo si verifica se l'offerta di volo non sia scaduta e se anche in questo caso l'esito è positivo si invia all'utente la conferma di accettazione dell'offerta. In caso contrario lo si informa dell'esito negativo dovuto alla scadenza dell'offerta o del token non valido ed il flusso termina con un errore.

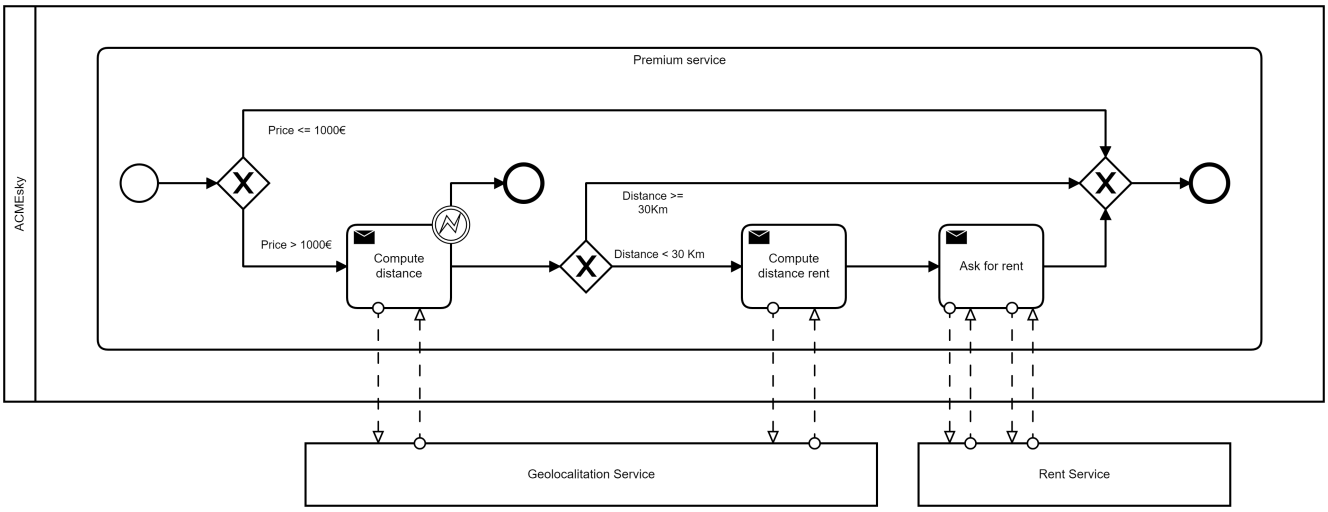
Pagamento dei voli



Il sottoprocesso inizia con la richiesta, da parte dell'utente, di pagamento del biglietto relativo all'offerta accettata. **ACMEsky** a questo punto si prende l'onere di prenotare i biglietti facendone richiesta all'**Airline Service** che fornisce i voli dell'offerta, la quale invierà in risposta i biglietti. Se c'è un errore relativo all'impossibilità di prenotare l'offerta poichè al servizio di airline risulta che l'offerta sia già stata acquistata o per qualsiasi altro problema, si invia un messaggio di errore all'utente ed il flusso termina.

Se la prenotazione va a buon fine, **ACMEsky** chiederà il link di pagamento a **Bank Service**, la quale glielo invierà in risposta a meno di errori nel processo. Successivamente il link viene inoltrato all'utente che procederà al pagamento sulla piattaforma di **Bank Service**. Infine, **Bank Service** comunica l'esito ad **ACMEsky** che proseguirà nel sottoprocesso dei servizi premium. Se il servizio della banca non risponde entro 5 minuti dalla generazione del link si procede alla compensazione dei biglietti e del pagamento, in via preventiva. In questo caso il processo termina con errore.

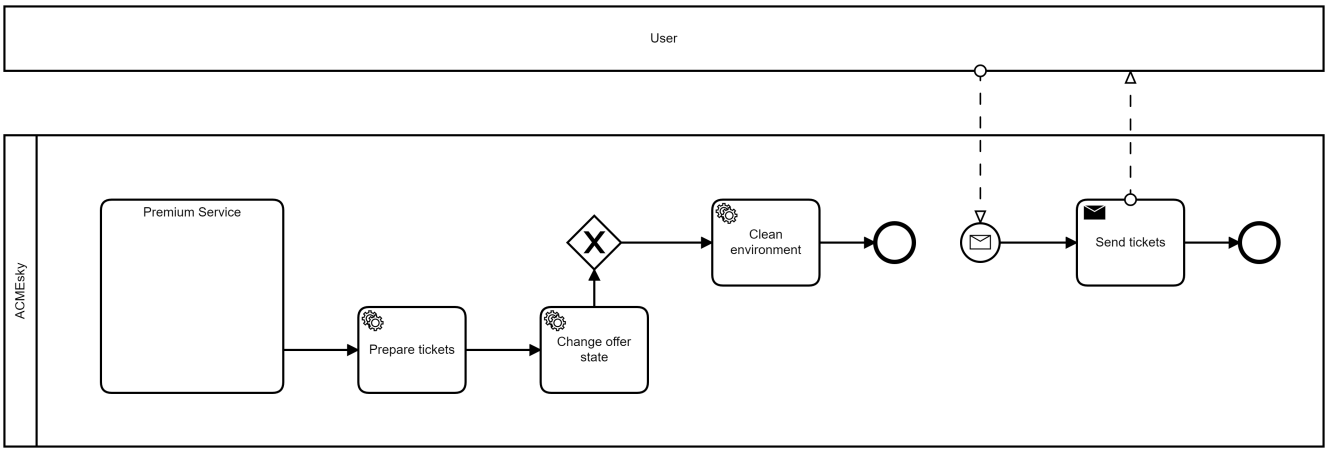
Servizi Premium voli



In questa fase, se vengono rispettate le condizioni, vengono applicati all'offerta i servizi premium. Inizialmente **ACMEsky** controlla il prezzo dell'offerta, se questo supera i mille euro invia una richiesta al servizio di **Geolocalizzazione** per calcolare la distanza dell'utente dall'areoporto. Nel caso in cui la distanza

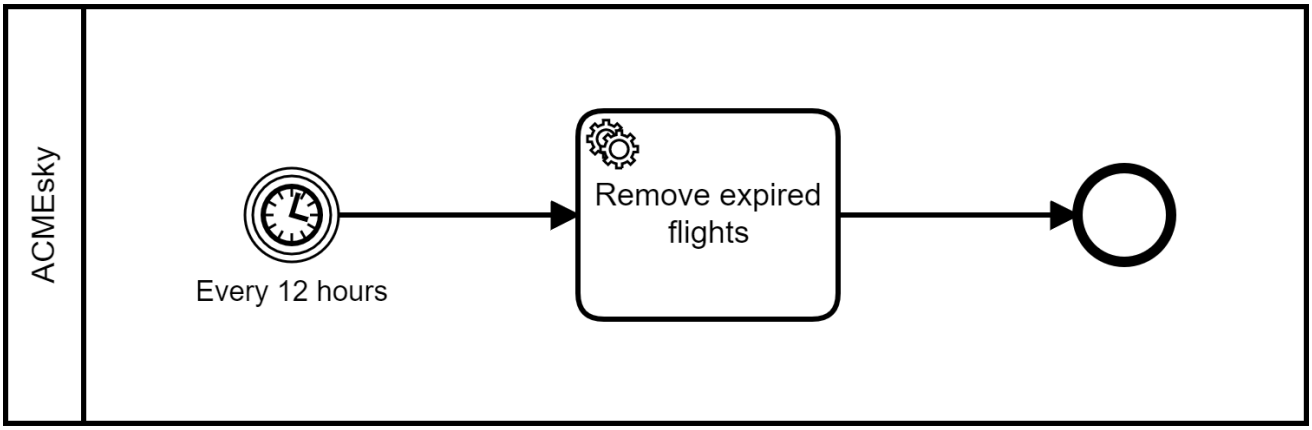
sia superiore ai 30 km si richiede al **Rent Service** più vicino se c'è la possibilità di offrire all'utente un trasferimento dal suo domicilio all'aereoporto. Questa operazione viene ripetuta sia all'andata che al ritorno, e in tal caso modificherà i biglietti includendo tutte le informazioni dei due trasferimenti. In caso la distanza sia inferiore ai 30Km o il prezzo dell'offerta sia inferiore a 1000€ non verrà richiesto nessun servizio.

Invio Biglietti



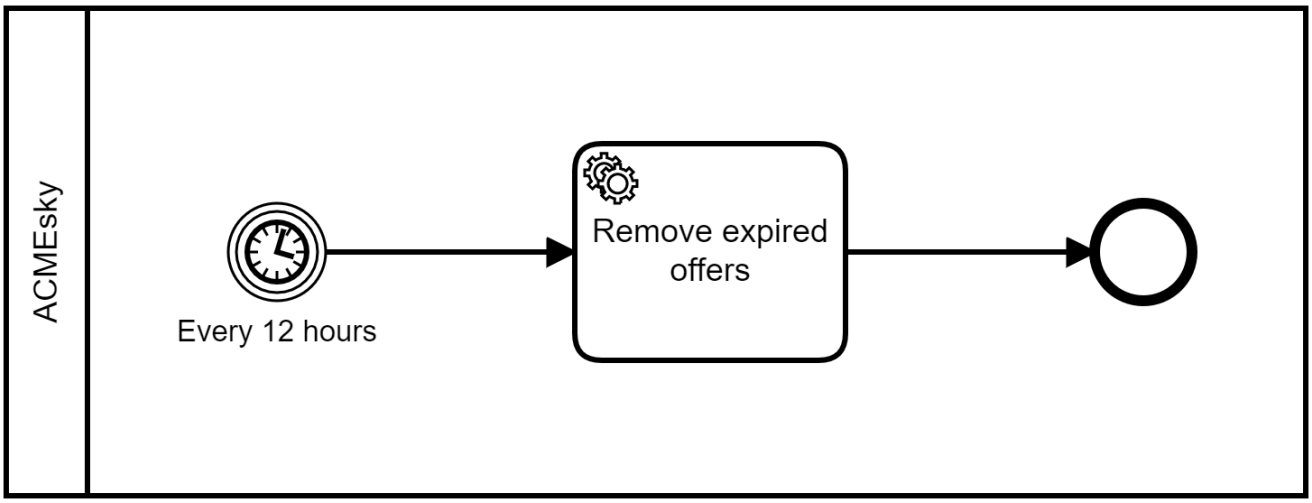
Arrivati a questo punto viene cambiato lo stato dell'offerta e viene preparato il pdf contenente i biglietti che l'utente potrà scaricare. L'utente può in qualunque momento richiedere i biglietti che ha acquistato.

Rimozione dei voli scaduti



Il processo di cancellazione dei voli scaduti presenti nel database avviene ogni 12 ore. I voli scaduti sono quei voli la cui data di scadenza è antecedente a quella in cui si effettua l'operazione di cancellazione. La cancellazione non comporta l'eliminazione effettiva del record che rappresenta quel volo, bensì un cambiamento di stato che porta ACMEsky a non considerare più quel volo come disponibile.

Rimozione delle offerte scadute

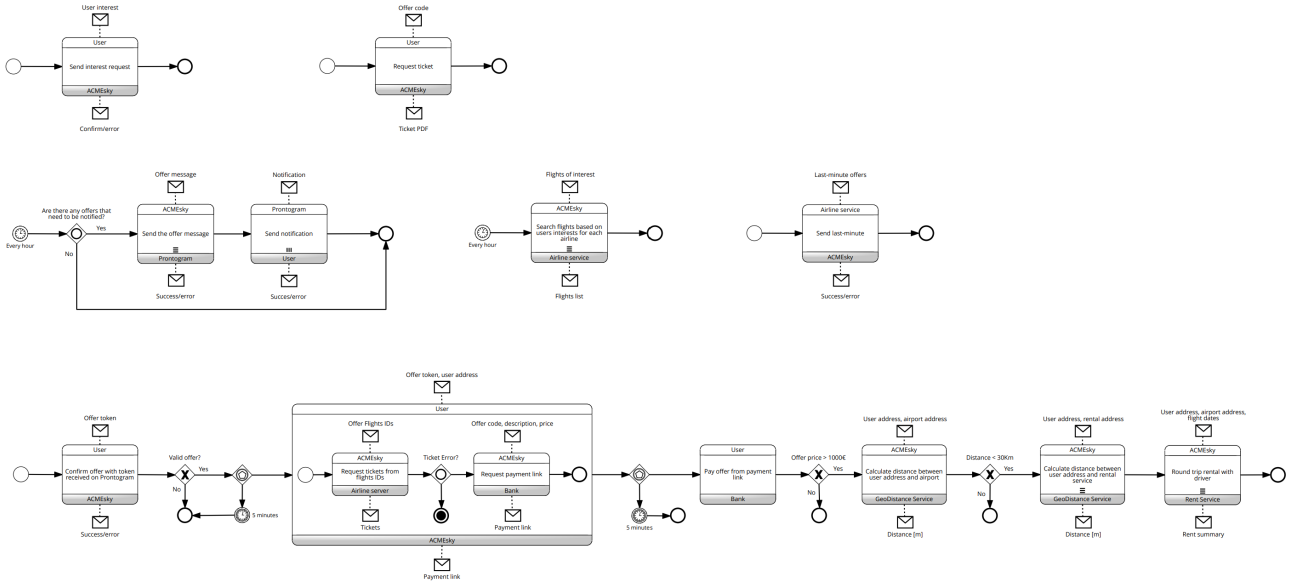


Il processo di cancellazione delle offerte scadute presenti nel Database avviene ogni 12 ore. Le offerte di volo scadute comprendono i voli di andata e ritorno. Le offerte scadute sono quelle la cui data di scadenza del volo di partenza è antecedente a quella in cui si effettua l'operazione di cancellazione. La cancellazione non comporta l'eliminazione effettiva del record, bensì, un cambiamento di stato che porta ACMEsky a non considerare più i voli dell'offerta (e l'offerta in sè) come disponibili.

Diagramma delle coreografie BPMN

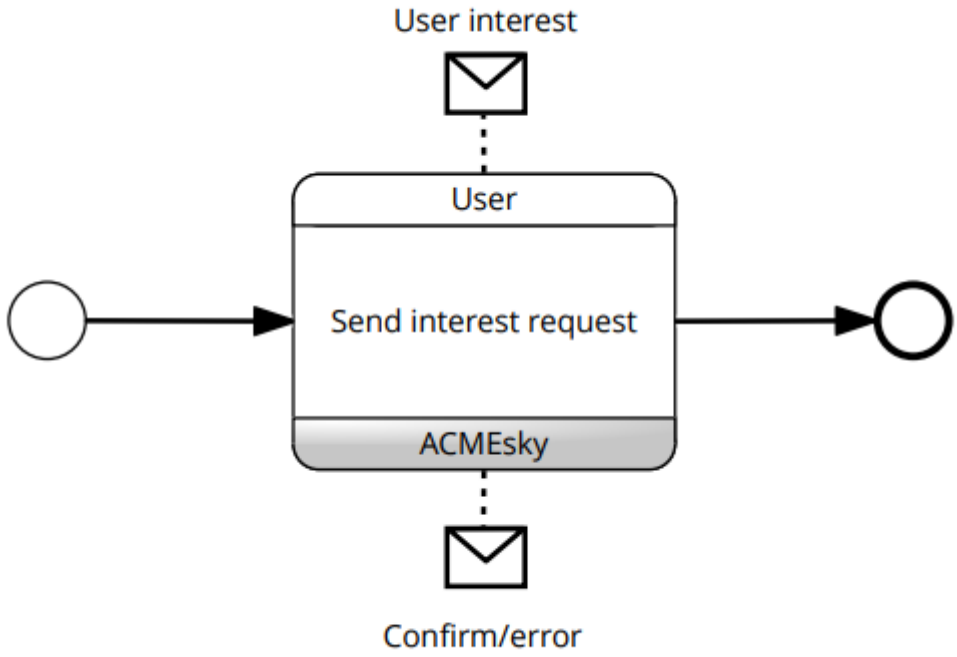
In questa sezione vengono mostrate le coreografie BPMN.

Diagramma completo



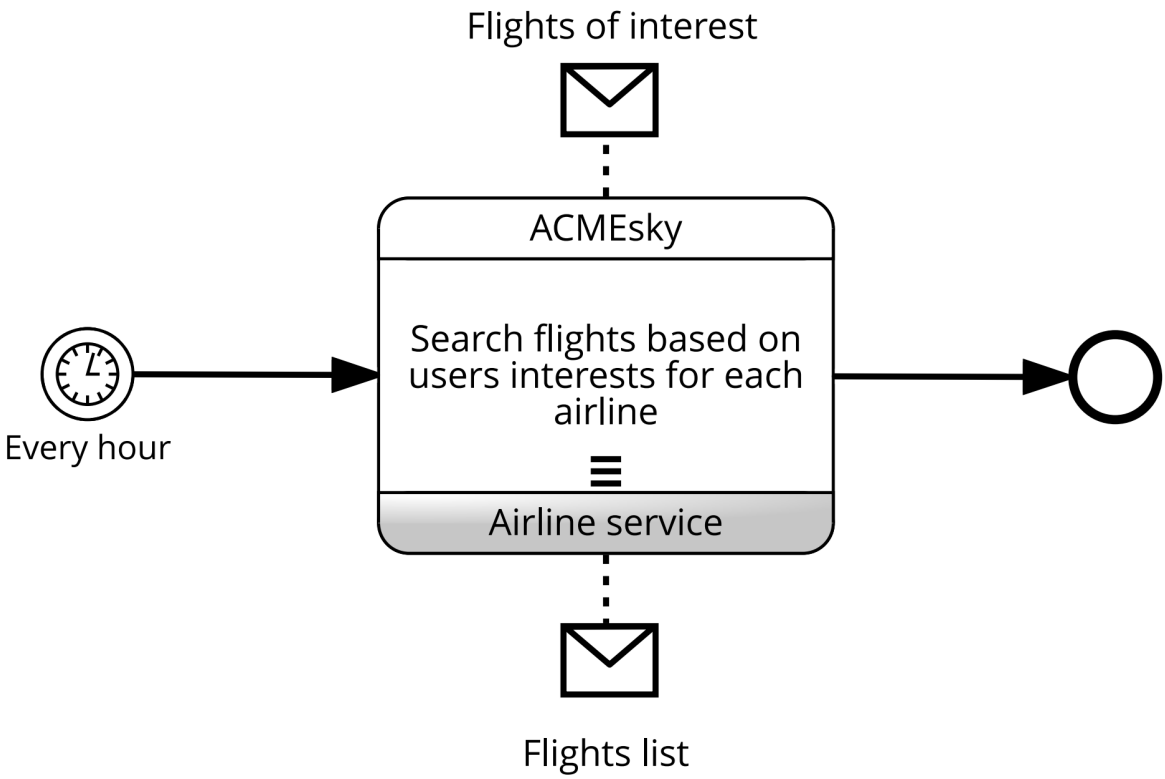
In questa sezione vengono mostrate le coreografie BPMN del sistema. Il diagramma è stato suddiviso in parti per poterle spiegare singolarmente.

Salvataggio degli interessi



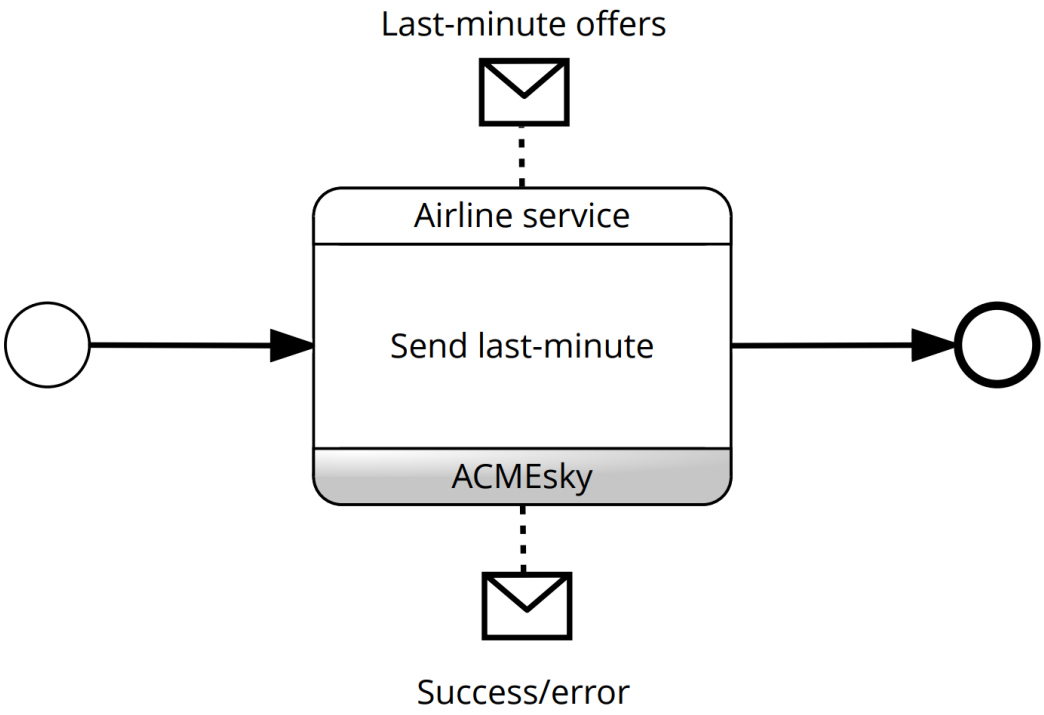
La coreografia descrive come l'utente e *ACMEsky* interagiscono nello scenario dell'invio degli interessi. L'utente manda una User Interest, mentre ACMEsky risponde con un messaggio di corretto inserimento o di errore nella richiesta.

Ricerca dei voli di interesse



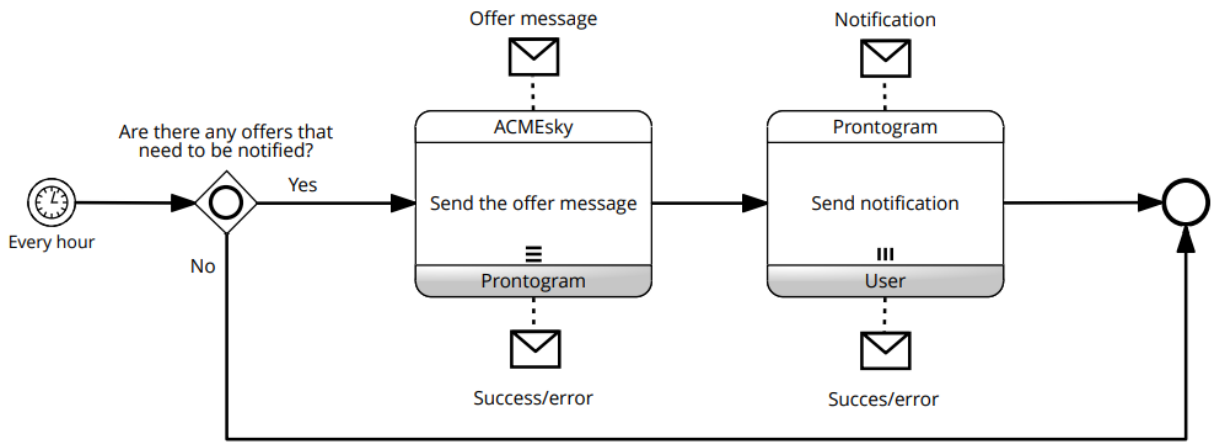
La coreografia descrive l'interazione tra *ACMEsky* e *Airline Service* al fine di cercare i voli che hanno una corrispondenza con quelli richiesti dagli utenti. Ogni ora e per ciascuna AirlineService registrata, *ACMEsky* recupera i voli di interesse degli utenti e effettua una richiesta all'*Airline Service* che restituisce la lista dei voli disponibili.

Registrazione dei voli last-minute



La coreografia descrive come *ACMEsky* e una compagnia aerea interagiscono nello scenario della ricezione e salvataggio dei voli last-minute. *Airline Service* invia i voli last-minute ad ACMEsky, che risponde con un messaggio di corretto inserimento o di errore nella richiesta.

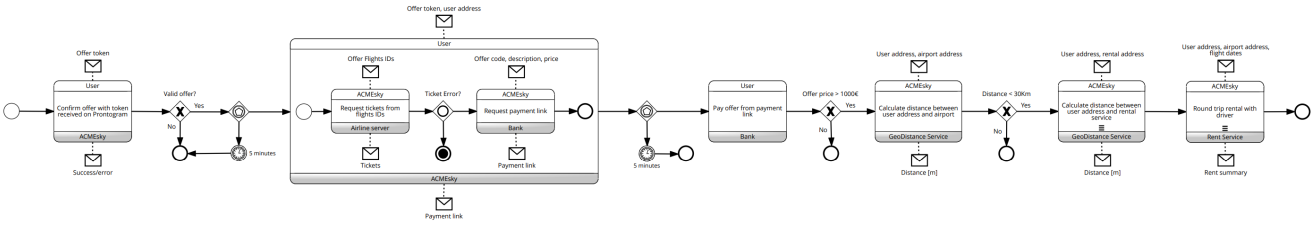
Generazione dell'offerta di volo



La coreografia mostra come *ACMEsky*, *Prontogram* e l'utente si relazionano al fine di notificare l'utente della corretta generazione dell'offerta. Ogni ora, e se ci sono delle offerte di volo appena generate (altrimenti il flusso termina), *ACMEsky* manda un messaggio a *Prontogram* che a sua volta risponde con un messaggio di

corretto inserimento o di errore nella richiesta. Successivamente in caso di esito positivo *Prontogram* manda una notifica all'utente.

Conferma e acquisto dell'offerta

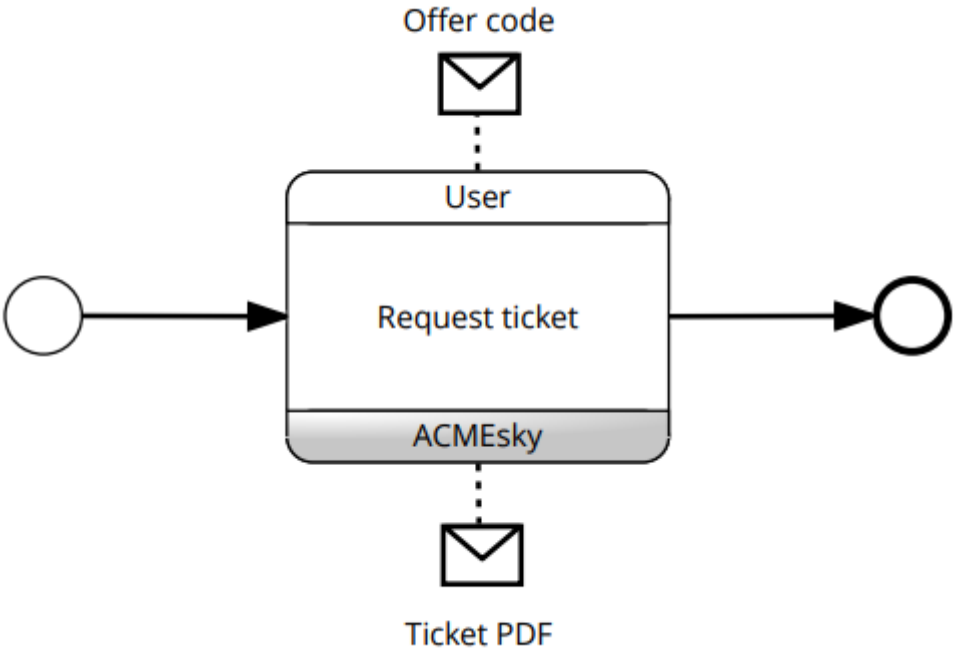


La coreografia descrive come l'utente, *ACMEsky*, *Bank Service*, *Airline Service*, *GeoDistance Service* e *Rent Service* interagiscono nel contesto della conferma e acquisto dell'offerta di volo da parte dell'utente. Quest'ultimo conferma la volontà di voler acquistare l'offerta proposta inserendo il token ricevuto sull'app di Prontogram ad *ACMEsky*, il quale risponde con un messaggio di corretto inserimento o di errore nella richiesta.

ACMEsky verifica che l'offerta sia ancora valida, se non lo è il flusso termina. Altrimenti, il processo continua fino alla sub-choreography, in cui l'utente passa il token dell'offerta ed il suo indirizzo per richiedere il pagamento, che deve svolgere entro cinque minuti, pena la fine del processo. *ACMEsky* recupera l'offerta corrispondente al token ed effettua una richiesta ad *Airline Service* che restituisce i biglietti. Se l'offerta è ancora disponibile per l'acquisto *ACMEsky* procede con il recupero del link del pagamento a *Bank Service* fornendo i dettagli dell'offerta. La banca restituirà ad ACMEsky il link, che successivamente verrà inviato all'utente. Se l'utente non effettua il pagamento tramite il link fornitogli entro cinque minuti il processo termina.

Se il prezzo dell'offerta è superiore a 1000€, *ACMEsky* calcolerà la distanza tra l'indirizzo dell'utente e quello dell'aereoporto di partenza inviando i rispettivi indirizzi al servizio di per l calcolo delle distanze che restituirà la distanza tra i due punti. Se questa è superiore ai 30Km, *ACMEsky* si servirà nuovamente di *GeoDistanceService* per trovare la compagnia di noleggio più vicina. Infine, viene prenotato il trasporto per l'andata e per il ritono.

Recupero del biglietto



La coreografia descrive come l'utente richiede ad *ACMEsky* il biglietto precedentemente acquistato. L'utente specifica il biglietto a cui è interessato inviando il codice dell'offerta ed *ACMEsky* recupera il biglietto corrispondente al codice in formato PDF.

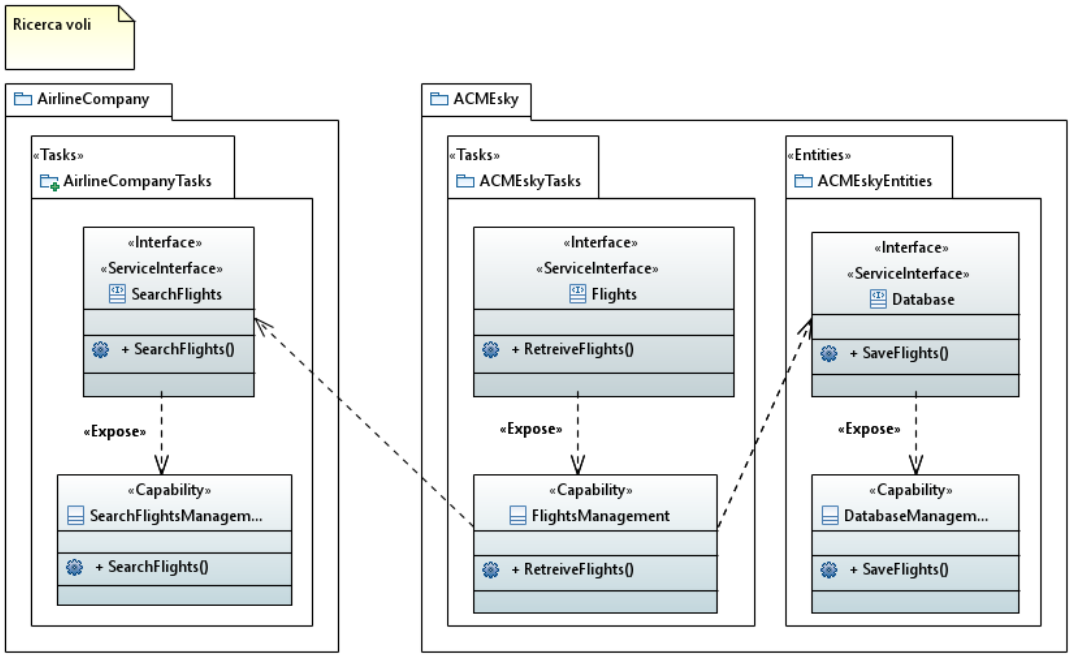
Diagrammi UML

In questa sezione della documentazione vengono mostrati i diagrammi UML, il cui scopo principale è quello di descrivere le interazioni che vi sono tra i vari servizi che fanno parte della SOA, attraverso l'utilizzo di ***capability*** e ***interface***. I diagrammi sono stati implementati utilizzando il profilo TinySOA.

Nei diagrammi UML sono riportate tre diverse tipologie di servizi:

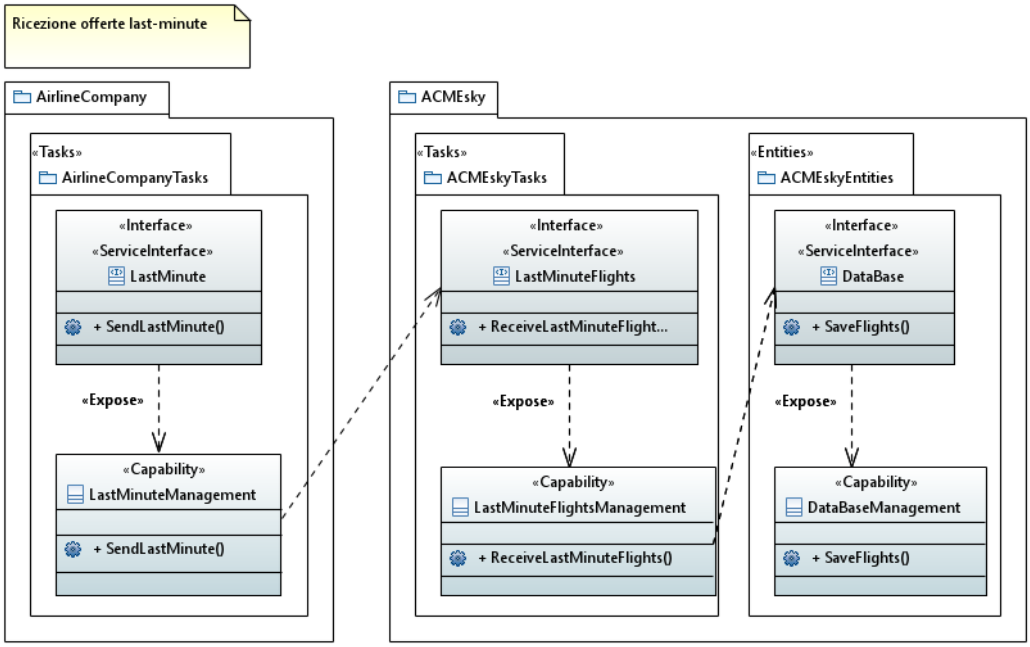
- **Task**: espone le *capability* facenti parte dei processi interni all'organizzazione;
- **Entity**: fa riferimento ad una singola attività, possibilmente automatizzata;
- **Utility**: sono simili ai task, però non appartengono al dominio del problema.

Richiesta voli



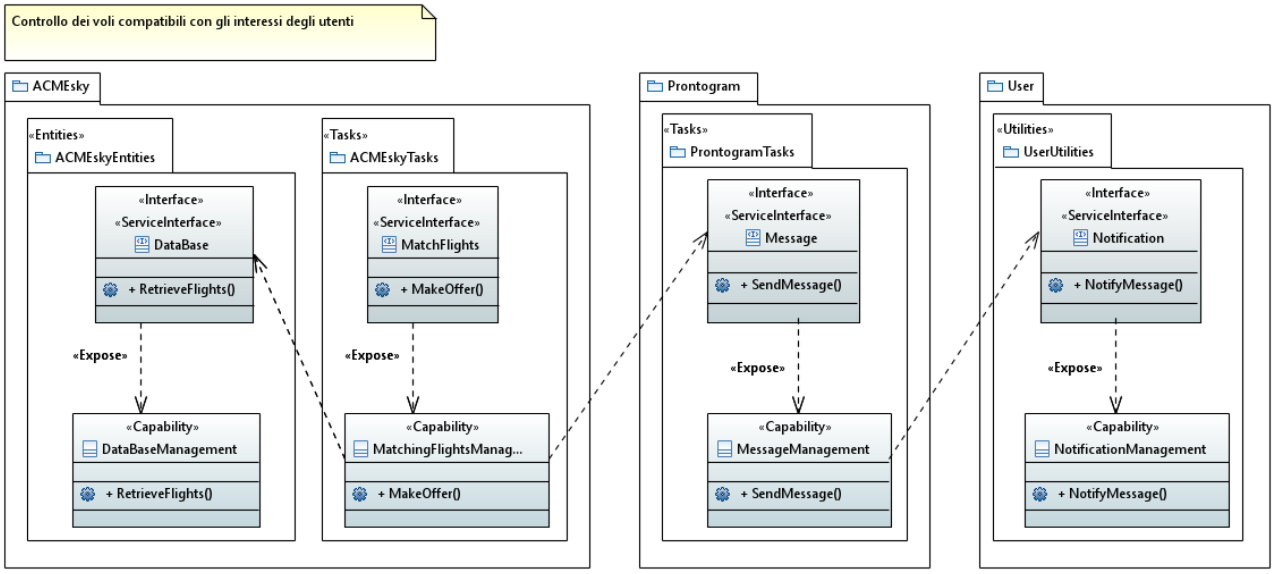
Nel diagramma riportato qui sopra vengono descritte le *capabilty* inerenti alla richiesta voli. In particolare per il ruolo di ACMESky vengono esposte le seguenti *capability*: *FlightsManagement* e *DatabaseManagement* le quali vengono esposte da due interfacce *Flights* e *Database*. La capability *FlightsManagement* ha lo scopo di interrogare e ricevere le offerte di voli dalle compagnie aeree. Mentre, la capability *DatabaseManagement* si occupa di salvare le offerte ricevute dalle compagnie aeree nella base dati.

Ricezione offerte last-minute



Nel diagramma riportato qui sopra vengono descritte le capabilty inerenti alla ricezione delle offerte last-minute. In particolare per il ruolo di ACMESky vengono esposte le seguenti *capability*: *LastminuteFlightsManagement* e *DatabaseManagement* le quali vengono esposte da due interfacce *LastMinuteFlights* e *DataBase*. La capability *FlightsManagement* ha lo scopo di ricevere le offerte dei voli last-minute dalle compagnie aeree. Quest'ultime verranno poi memorizzate nella base dati attraverso la capability *DatabaseManagement*.

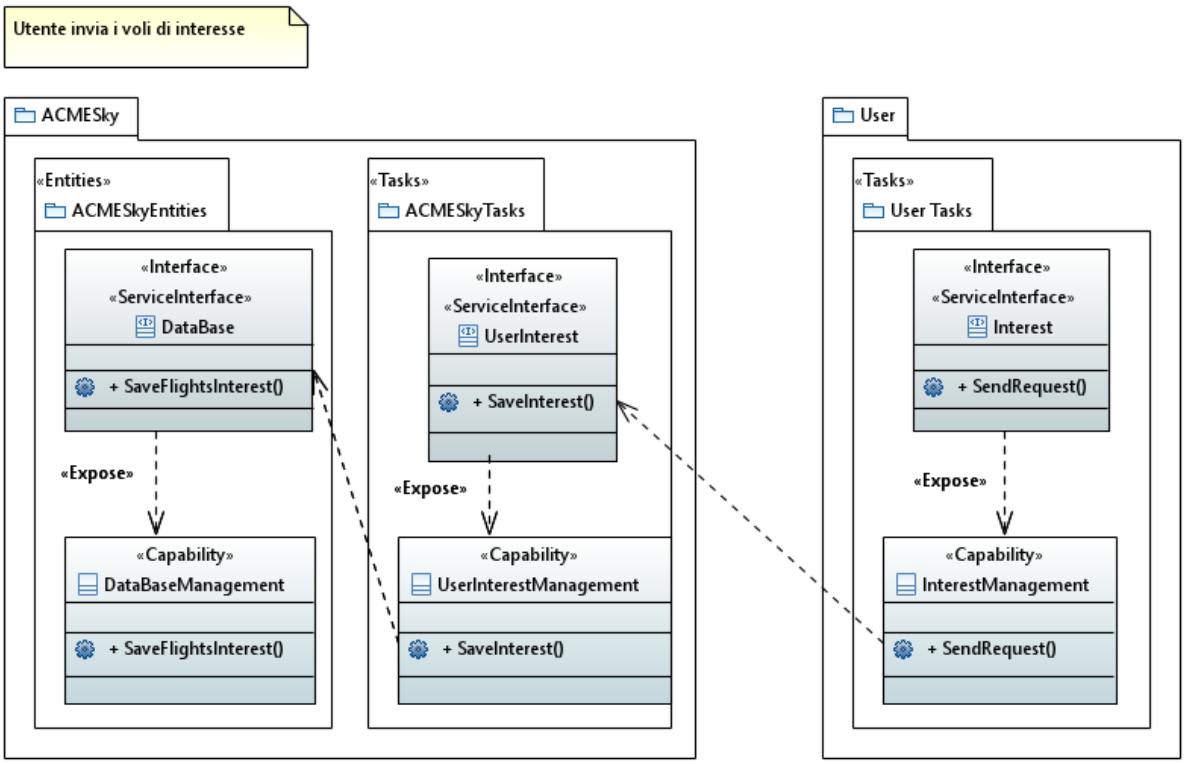
Controllo periodico dei voli compatibili con gli interessi degli utenti



Nel diagramma riportato qui sopra vengono descritte le capabilty inerenti alla ricezione delle offerte last-minute. In particolare per il ruolo di ACMESky vengono esposte le seguenti *capability*: *MatchingFlightsManagement* e *DatabaseManagement* le quali vengono esposte da due interfacce *HourlyMatchFlights* e *DataBase*. La capability *MatchingFlightsManagement*, dopo avere reperito i voli dalla

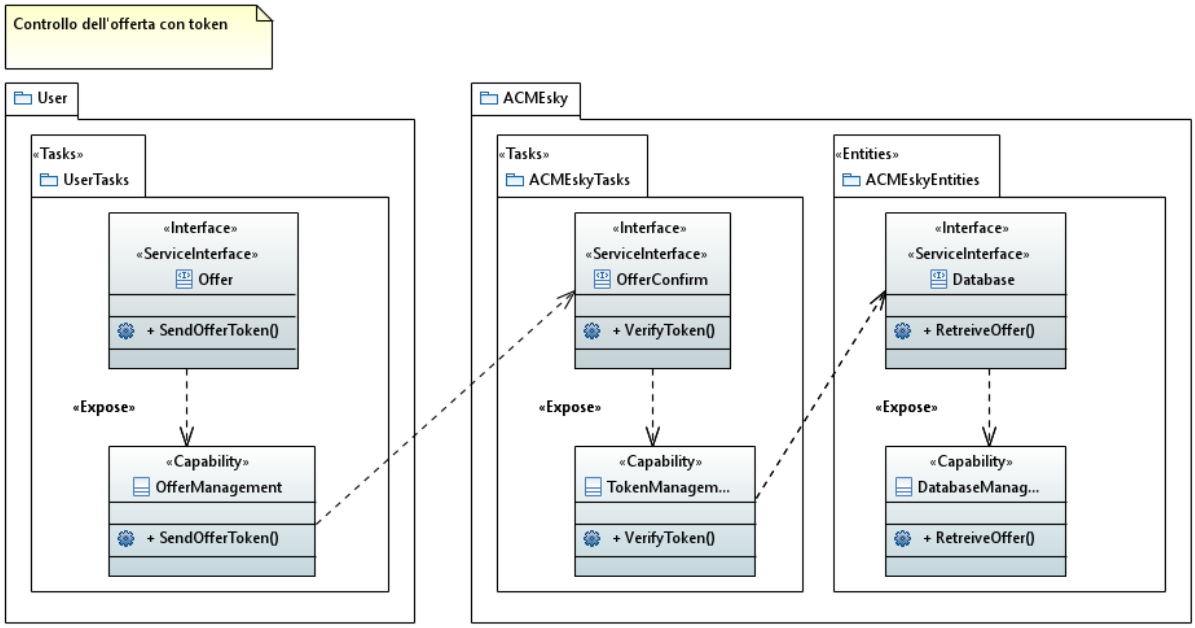
base dati attraverso *DatabaseManagement*, ha lo scopo di trovare l'offerta compatibile con quella di interesse per l'utente. Inoltre la capability *MatchingFlightsManagement* dipende dalla capability di invio messaggi di ProntoGram, *MessageManagement* esposta dalla relativa interfaccia: *Message*. Per contattare l'utente la capability di ProntoGram (*MessageManagement*) dipende da un'altra capability *NotifyMessage* la quale ha lo scopo di segnalare all'utente la presenza di nuovi voli attraverso un messaggio di notifica.

Invio richiesta del volo da parte dell'utente



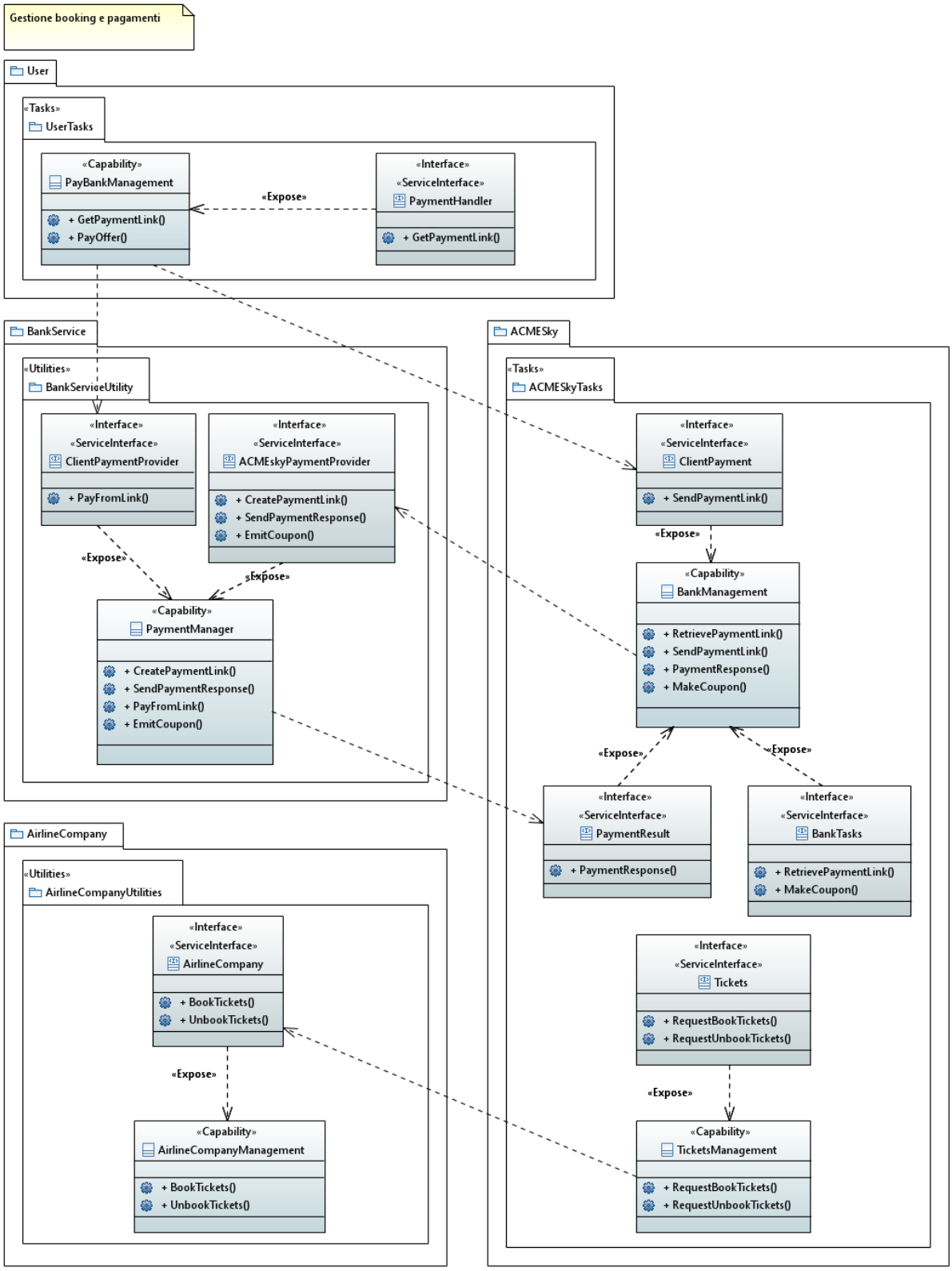
Nel diagramma riportato qui sopra vengono descritte le capabilty inerenti all'invio da parte di un utente della richiesta di un volo. In particolare per il ruolo di ACMESky vengono esposte le seguenti *capability*: *UserInterestManagement* e *DatabaseManagement* le quali vengono esposte da due interfacce rispettivamente *UserInterest* e *DataBase*. In particolare la capability *UserInterestManagement* si occupa di ricevere la richiesta di un volo da parte di un utente esponendo quindi una dipendenza per la capability *UserRequestManagment*, la quale ha lo scopo di inviare la richiesta dell'utente. Invece, la capability *DatabaseManagement* si occupa di salvare la richiesta nella base dati di ACMESky.

Controllo dell'offerta con token



Nel diagramma riportato qui sopra vengono descritte le capabilty inerenti al controllo dell'offerta di interesse attraverso l'utilizzo del token inviato dall'utente. In particolare per il ruolo di ACMESky vengono esposte le seguenti *capability*: *TokenManagement* e *DatabaseManagement* le quali vengono esposte da due interfacce rispettivamente *ReceiveToken* e *DataBase*. Queste capability permettono al sistema di ricevere il token da parte di un utente, verificare la correttezza del token ricevuto e reperire la relativa offerta dalla base dati di ACMESky. La capability *TokenManagement* dipende dalle interfacce che espongono la capability *TokenManagement* dell'utente per poter verificare la validità del codice inserito.

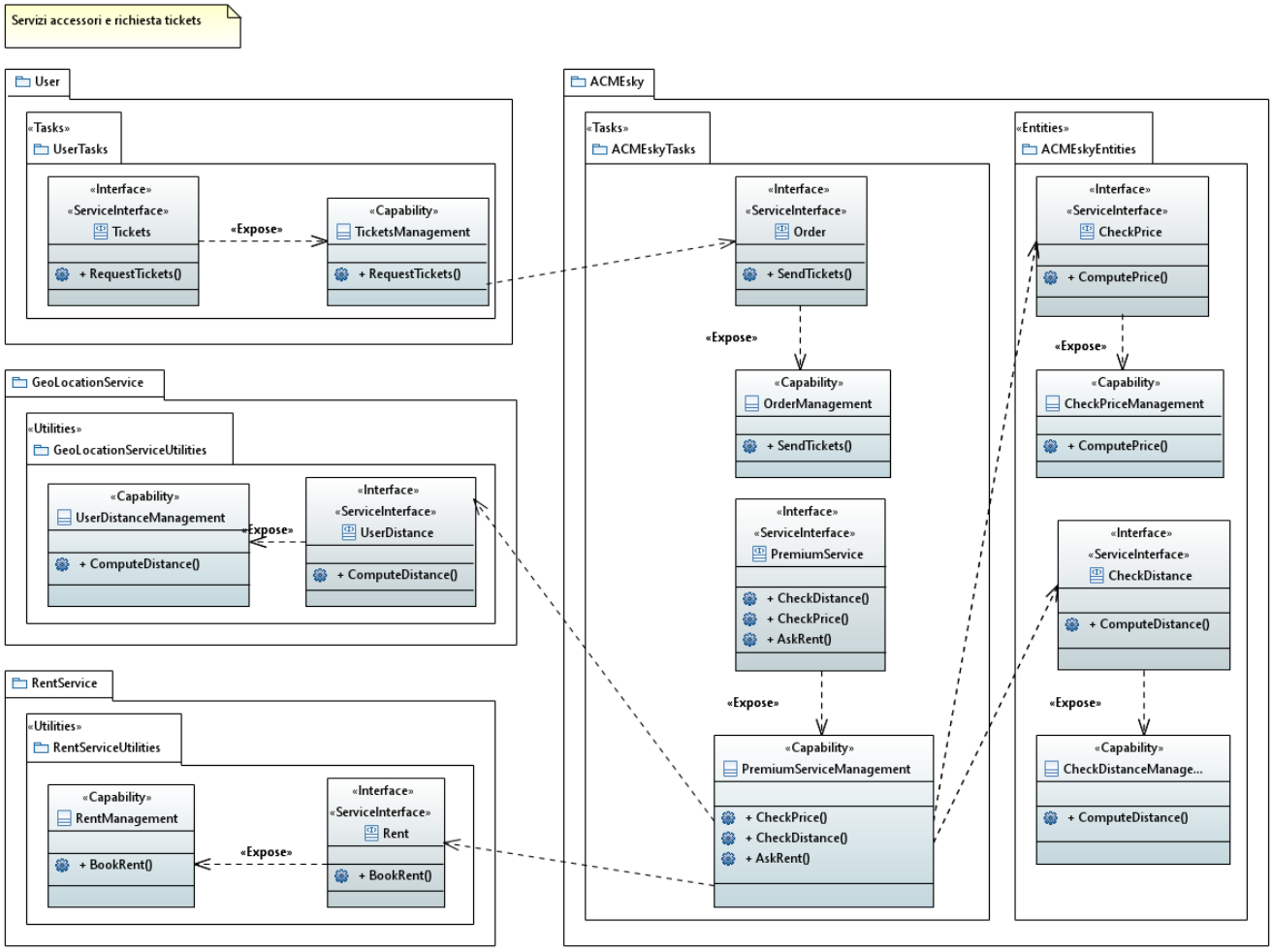
Gestione dei pagamenti



Nel diagramma riportato qui sopra vengono descritte le capabilty inerenti alla gestione dei pagamenti effettuati dall'utente indirizzati ai vari servizi. In particolare per il ruolo di ACMESky vengono esposte le seguenti *capability*: *BankManagement* e *ArlineCompanyManagement* le quali vengono esposte da un'unica interfaccia *Bank* e *AirlineCompany*. La capability *BankManagement* di ACMESky si occupa di ricevere il link di pagamento dalla banca e di inviarlo all'utente. Quest'ultima riceve anche tutte le informazioni relative ad

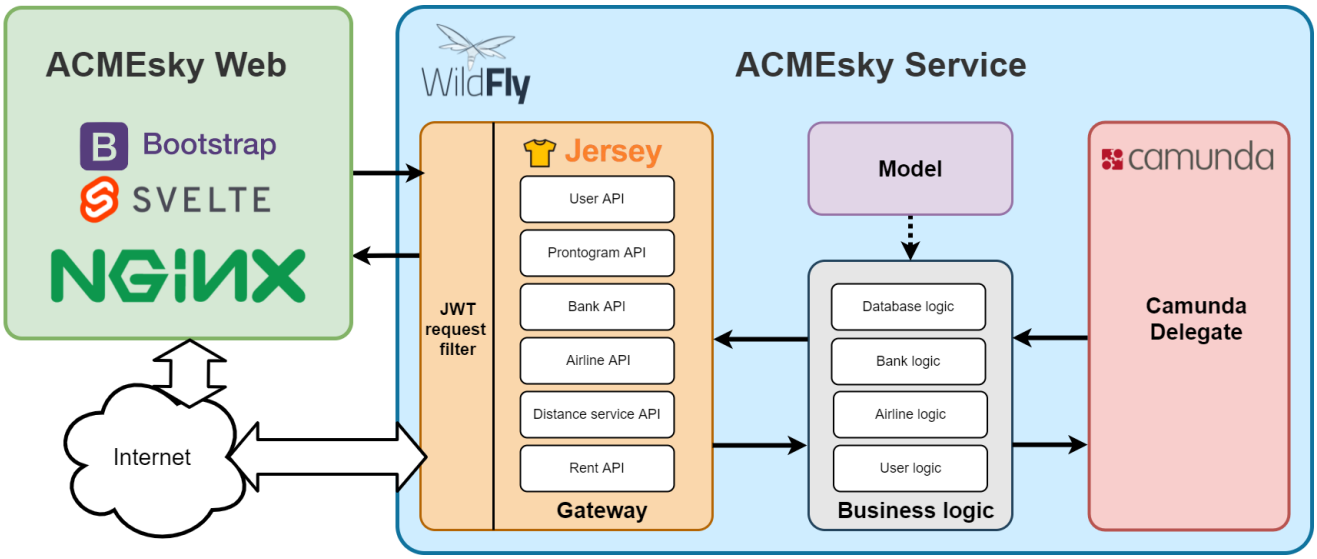
un avvenuto pagamento (successo o insuccesso). La capability *ArlineCompanyManagement* si occupa di prenotare il volo di interesse o di eliminare la prenotazione in caso di errori nei sotto-processi.

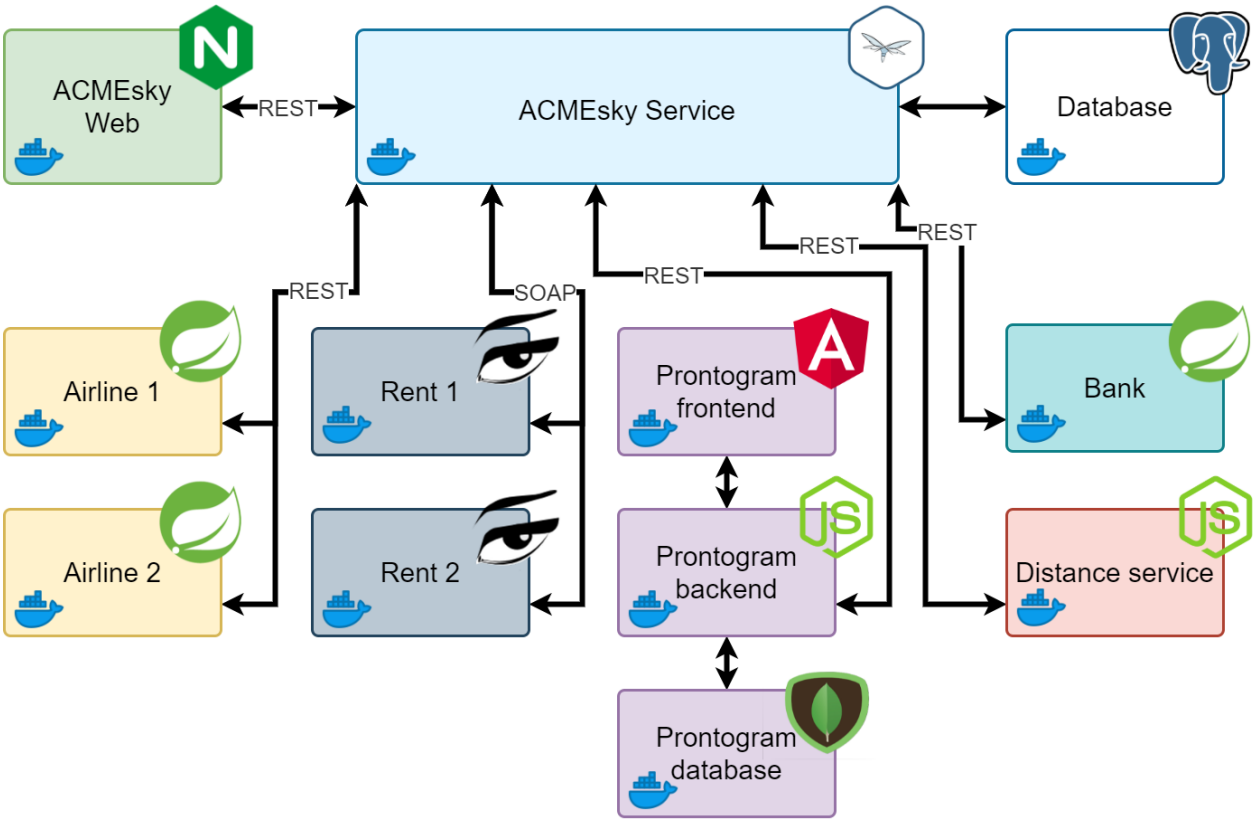
Servizi accessori e ricezione tickets



Nel diagramma riportato qui sopra vengono descritte le capabilty inerenti servizi accessori e ricezione dei tickets di interesse. In particolare per il ruolo di ACMESky vengono esposte le seguenti *capability*: *PremiumServiceManagement* e *OrderManagement* le quali vengono esposte da un'unica interfaccia *PremiumService* e *Order*. La capability *PremiumServiceManagement* si occupa di controllare tutte le condizioni per poter attivare il servizio premium, pertanto controlla se il prezzo dei tickets, la distanza dall'aeroporto e l'eventuale prenotazione di un servizio di trasporto. Infine la capability *OrderManagement* ha il compito di inviare i tickets all'utente.

Implementazione
In questa sezione si presenta la struttura del progetto, specificando per ciascun servizio le tecnologie impiegate e la struttura scelta per l'implementazione.





ACMEsky

- ACMEskyWeb: servizio sviluppato con SvelteJS, framework opensource per il frontend

- ACMEskyService: sviluppato utilizzando Java Enterprise Edition che implementa la specifica JAX-RS (Java API for RESTful Web Services), un set di interfacce e annotazioni che facilitano lo sviluppo di applicazioni lato server. Per quanto riguarda il deployment si è scelto Red Hat JBOSS Enterprise Application Platform che offre supporto completo a Java EE in tutti gli ambienti applicativi. Questa scelta è dovuta anche dal fatto che una delle distribuzioni di Camunda Platform si basa su JBOSS, oltre al fatto che la distribuzione basata su Apache potrebbe dare problemi Camunda è un Java-based framework che supporta BPMN 2.0 per il workflow e l'automazione dei processi a livello di produzione. La distribuzione di Camunda si trova su un container Docker sul quale si carica il file .war del servizio in modo da poter automatizzare i processi. Java EE consente di creare dei file .war molto leggeri escludendo le librerie necessarie che l'application server su cui viene eseguito dovrà recuperare. Ciò aumenta la flessibilità e la manutenibilità del codice, infatti un'implementazione personalizzata come RESTEasy è meno flessibile nel caso in cui si passasse ad un altro deployment. Questa parte di ACMEsky si occupa di gestire le interazioni tra i vari servizi al fine di cercare e ottenere i voli dagli AirlineService, gestire le offerte con l'utente per mezzo del frontend, gestire il database, proporre all'utente le offerte basate sui suoi voli di interesse, procedere con la prenotazione, l'acquisto interagendo con il servizio bancario, l'applicazione di servizi aggiuntivi comunicando con i servizi di geolocalizzazione e di rent, e la preparazione dei biglietti.
- ACMEskyDB: la parte di ACMEsky utilizzata per la creazione, gestione ed eliminazione del Database utilizzato da ACMEskyService. Come Database si è scelto il DB gratuito ed open source di PostgreSQL, invece per la sua gestione si è scelto il tool di pgAdmin 4, il management tool open source di PostgreSQL. Per rendere il tutto più portatile ed indipendente possibili si custodisce il Database e pgAdmin 4 su due container Docker.

AirlineService(s)

Il servizio che simula delle semplici compagnie aeree. Sono state sviluppate con Spring Boot, framework di Java per realizzare applicazioni web e servizi REST con facilità. Per creare e gestire più servizi di compagnia aerea si è scelto di utilizzare Docker, il quale consente di creare e gestire simultaneamente più istanze di un'applicazione attraverso il file di docker-compose. Al momento i servizi di Airline Service sono due: uno che offre voli nazionali e uno che offre voli internazionali, ma modificando il docker-compose è possibile aggiungerne di nuovi o rimuovere quelli già presenti.

BankService

GeographicalDistanceService

Prontogram

Web-application che permette all'utente di ricevere le notifiche delle offerte dei voli dai servizi di ACMEsky. L'aaplicativo è stato sviluppato in

RentService(s)

Istruzioni per l'esecuzione del progetto

Prerequisiti

- Docker
- Docker compose
- Un po' di pazienza!

Esecuzione

Dalla root del progetto eseguire

```
docker-compose up --build
```

I singoli servizi non devono essere compilati perchè questa operazione viene già fatta all'interno dei container.

ACMEsky
ACMEskyService, il servizio principale del progetto

comando per fare la build

```
mvn package
```

comando per fare la build del servizio e far partire il container Docker

```
docker-compose up --build
```

Documentazione

ACMEskyService è il servizio principale di ACMEsky che si relaziona con i vari servizi (ACMEskyWeb, AirlineServices, RentServices, Prontogram, BankService,...) al fine di consentire agli utenti di prenotare ed acquistare le offerte di volo che desiderano.

ACMEskyService comunica con il servizio di ACMEskyDB (il quale consiste nel Database di PostgreSQL e del tool di pgAdmin4) per registrare gli utenti e i vari servizi che comunicano con ACMEsky, in modo da poterli riconoscere quando inviano richieste HTTP, anche per salvare le offerte di volo richieste dagli utenti (e i rispettivi voli), i voli delle compagnie aeree (last-minute e non), tutti gli aereoporti mondiali in codice IATA e le offerte di volo generate dai voli acquisiti dai servizi di volo secondo le offerte di interesse degli utenti. Inoltre interagisce con ACMEskyWeb per interfacciarsi con gli utenti: infatti raccoglie le offerte di volo che ciascun utente desidera, al momento dell'accettazione dell'inserimento del token per acquistare l'offerta verifica dal token se l'offerta è ancora valida e procede con il pagamento e del risultato dell'azione (restituzione del biglietto o cancellazione della prenotazione e rimborso). Infine il servizio di ACMEsky consentirà all'utente di visualizzare i biglietti dei voli precedentemente acquistati dopo aver applicato eventuali servizi aggiuntivi. ACMEskyService si relaziona con i servizi di AirlineService (compagnie aeree) al fine di cercare, tra i voli che offrono, quelli corrispondenti a voli di interesse degli utenti al fine di proporre loro offerte da acquistare, ricevere voli last-minute inviati dai servizi di volo con una certa frequenza, recuperare i biglietti dei voli che gli utenti vogliono acquistare tramite una richiesta HTTP alla compagnia aerea che offre i voli in questione (se non sono già stati prenotati in precedenza), e nel caso di errori o problemi di pagamento cancellare la prenotazione informando l'AirlineService corrispondente. Per quanto riguarda il pagamento, il servizio interroga BankService, il servizio bancario, per richiedere il link di pagamento da inviare all'utente tramite il servizio di ACMEskyWeb, il quale poi interagirà con la banca per gestire il pagamento. Infine BankService informerà ACMEsky solo se l'esito è positivo (in caso contrario quest'ultimo dovrà occuparsi della compensazione). Il servizio interagisce con il servizio di geolocalizzazione "GeolocalizationService" e con quelli dei servizi di noleggio "RentService" per applicare eventuali servizi aggiuntivi all'offerta di volo acquistata dell'utente. Al servizio di geolocalizzazione fa una richiesta HTTP per conoscere le posizioni dell'aereoporto del volo e dell'indirizzo dell'utente per calcolarne

la distanza, mentre ai servizi di noleggio chiede se è possibile accompagnare i passeggeri dal loro indirizzo all'aereoporto con un mezzo. Infine ACMEsky aggiunge i dettagli precedentemente descritti.

Panoramica

Struttura del servizio e tecnologie utilizzate

Il servizio è stato sviluppato utilizzando Java Enterprise Edition, il quale implementa la specifica JAX-RS (Java API for RESTful Web Services), un set di interfacce e annotazioni che facilitano lo sviluppo di applicazioni lato server. Per quanto riguarda il deployment si è scelto Red Hat JBOSS Enterprise Application Platform che offre supporto completo a Java EE in tutti gli ambienti applicativi. Questa scelta è dovuta anche dal fatto che una delle distribuzioni di Camunda Platform si basa su Wildfly, l'application server di JBOSS. Si è scelto inoltre di fornire un'interfaccia web per presentare le risorse di ACMEskyService richiamabili da servizi esterni tramite Swagger UI, un linguaggio di descrizione delle API RESTful che implementa la specifica di OpenAPI. I biglietti in formato pdf vengono generati grazie al framework opensource di Itext, che consente di convertire file html in pdf automaticamente. La build e l'esecuzione del servizio viene svolta tramite Docker, in particolare nel file docker-compose viene definito l'immagine del container da creare e la connessione con la rete "acmesky-net" che consente agli altri container esterni al progetto ma connessi alla rete di comunicare con ACMEskyService (db, compagnie aeree, banche, ecc.). Il progetto è composto dalle seguenti cartelle:

- camunda: questa parte del progetto si compone di tre sottocartelle, ovvero flights_manager, offers_manager e user_manager: questo perchè si è voluto riprendere la struttura del diagramma BPMN Total, che divide i vari flussi di esecuzione dei processi in queste tre pool a seconda della loro funzione e degli attori con cui interagiscono. Ciascuna classe implementa rappresenta un task di un processo o sottoprocesso presente nel flusso di esecuzione. La classe implementa l'interfaccia JavaDelegate e definisce il metodo execute, il quale viene richiamato ogni volta che si fa partire il flusso del diagramma a cui appartiene. La cartella flights_manager si suddivide nelle successive cartelle: last_minute che ospita al proprio interno un file che consente di salvare le offerte che le compagniee aeree inviano ad ACMEsky, remove_expired_flights la quale include il file RemoveExpiredFlights che rimuove le i voli scaduti ricevuti dai servizi di volo, e search_flights a sua volta composto da quattro file la cui esecuzione consente di recuperare la lista degli AirlineService, recuperare la lista dei voli da cercare, fare una richiesta alle risorse dei servizi per richiedere i voli corrispondenti ai voli cercati e salvare tali voli. La cartella offers_manager si compone delle seguenti parti: remove_expired_offers, ovvero la cartella contentente il file che una volta eseguito rimuove le offerte di volo e relativi voli scaduti, e i restanti file i cui metodi una volta richiamati i task corrispondenti permettono di controllare i voli di interesse disponibili, scegliere quindi tra i voli disponibili quelli che corrispondono agli interessi degli utenti, preparare l'offerta e inviarla all'utente che ha richiesto quei voli. La cartella user_manager si suddivide in: book_payment, che riguarda la prenotazione e acquisto dei biglietti aerei, confirm_offer, il cui contenuto si occupa della conferma dell'offerta espressa dall'utente inserendo il token e controllando che l'offerta sia ancora valida, premium_service, per quanto concerne l'applicazione di eventuali servizi aggiuntivi all'offerta, save_interest include file collegati ai processi e sottoprocessi che si occupano del salvataggio dell'offerta di interesse degli utenti, gli altri file sono legati ai task finali che

riguardano il cambiamento dello stato dell'offerta acquistata con successo, la cancellazione del contenuto delle variabili dell'ambiente e l'invio del biglietto acquistato dall'utente quando lo richiede. I file della folder utils elencano tutte le variabili degli eventi e degli eventi di errore, stabiliscono le variabili dei processi e le caratteristiche lo stato dei processi stessi, incluso come impostarne lo stato, recuperarlo o recuperarlo per l'ultima volta prima di cancellarli.

- gateway: questa parte del progetto si compone di una cartella per ciascun servizio, in cui si descrivono le route accessibili dai servizi esterni per registrarsi per la prima volta, autenticarsi e interagire con ACMEsky per raggiungere i loro obiettivi. Descrive inoltre le richieste che il servizio effettua verso gli attori esterni che verranno utilizzate dalle istanze manager della logica. Ovviamente include anche i Data Transfer Object necessari per fare le richieste e per chiamare le risorse.

- logic: ospita i manager che utilizzano i modelli definiti dal model, le risorse e le chiamate stabilite dal gateway per interagire con il Database e con i vari servizi esterni. In particolare il DatabaseManager si occupa delle query sul DB, AirlineManager si occupa di tutto ciò che riguarda le compagnie aeree, BankManager per quanto riguarda le banche, InterestManager e OfferManager gestiscono rispettivamente gli interessi degli utenti e le offerte di volo generate dal servizio e quelle di interesse degli utenti.

- model: descrive le entità coinvolte nel progetto, ovvero i servizi esterni, i voli, le offerte generate e gli interessi degli utenti e i relativi voli. Specifica gli attributi degli oggetti istanziati dalle classi, le rispettive proprietà nelle tabelle relazionali del database autogenerate a partire dalle classi stesse e le relazioni che intercorrono tra i campi delle tabelle.

- security: si occupa dell'autenticazione dei servizi che fanno richieste attraverso le route di ACMEsky controllandone le credenziali, autorizza questi servizi restituendo, nel caso l'autenticazione abbia avuto successo, un token con il ruolo corrispondente a quello con cui si sono registrati e una scadenza. Al momento della chiamata ad una risorsa verifica che il token presente nell'header non sia scaduto e che il ruolo del jwt corrisponda sia a quello presente nel proprio record in DB e sia a quello richiesto dalla route. Nel caso positivo la richiesta del servizio viene accettata, altrimenti viene rigettata. Non mancano gli header da aggiungere alle richieste per il CORS e le costanti per fare i jwt token e i ruoli dei servizi

- utils: contiene i file che descrivono le possibili stringhe di errore, alcune variabili d'ambiente come la finestra di tempo massimo in cui è possibile prenotare un volo d'andata a partire dalla data odierna e la classe che si occupa di creare il pdf dei biglietti.

Risorse necessarie per tutti i servizi

Risorsa	Descrizione
POST/ auth/	Questa risorsa consente ai servizi che vogliono chiamare le route del servizio di autenticarsi. Si richiede un oggetto AuthRequestDTO come parametro, il quale è composto da un attributo username e un attributo password. Il servizio chiamante dovrà inserire nel body l'argomento in formato JSON e ACMEsky restituirà un token

	valido per tutte le richieste alle route che richiedono il ruolo codificate in esso, a patto che le credenziali inserite siano corrette.
PUT/ auth/refresh	Questa risorsa consente ai servizi BankService, ACMEskyWeb, AirlineService di richiedere un nuovo token a partire da quello scaduto.

Risorse per AirlineServices

Risorsa	Descrizione
POST/ airlines/last_minute	Questa risorsa è riservata esclusivamente agli AirlineServices, quindi i servizi dovranno presentare un bearer token valido rilasciato da ACMEskyService con una richiesta alla risorsa di autenticazione. La chiamata a questa risorsa richiede come parametro una lista di oggetti AirlineFlightOfferDTO che verrà passata ad un metodo che avvierà un task di Camunda che si occuperà di convertire la lista di DTO in oggetti Flight da salvare in DB.

Risorse per BankServices

Risorsa	Descrizione
GET/ bank/confirmPayment	Questa risorsa è riservata esclusivamente ai servizi BankServices, quindi essi dovranno presentare un bearer token valido rilasciato da ACMEskyService con una richiesta alla route di autenticazione. La chiamata a questa risorsa richiede come parametro il token dell'offerta di volo acquistata dall'utente per informare ACMEskyService del fatto che l'offerta è stata acquistata.

Risorse per ACMEskyWeb

Risorsa	Descrizione
GET/ airports/	Questa risorsa è riservata esclusivamente al servizio ACMEskyWeb, che dovrà presentare un bearer token valido rilasciato da ACMEskyService con una richiesta alla risorsa di autenticazione. Il servizio web di ACMEsky chiama questa risorsa passando la query di ricerca dell'aereoporto per recuperare la lista dei suggerimenti per gli aereoporti di partenza e di arrivo dei voli sulla base dei caratteri inseriti dagli utenti nei rispettivi form della pagina al momento della ricerca.
GET/ airports/{code}	Questa risorsa è riservata esclusivamente al servizio ACMEskyWeb, che dovrà presentare un bearer token valido rilasciato da ACMEskyService con una richiesta alla risorsa di autenticazione. Il servizio web di ACMEsky chiama

	questa risorsa passando la query di ricerca dell'aereoporto per recuperare l'aereoporto di partenza e di arrivo dei voli sulla base dei codici aereoportuali inseriti dagli utenti nei rispettivi form della pagina.
POST/ interests/	Questa risorsa è riservata esclusivamente al servizio ACMEskyWeb, che dovrà presentare un bearer token valido rilasciato da ACMEskyService con una richiesta alla risorsa di autenticazione. Il servizio web di ACMEsky chiama questa risorsa con un oggetto DTO dell'offerta di volo di interesse dell'utente per consentire ad ACMEskyService di registrare l'interesse dell'utente in DB.
GET/ interests/	Questa risorsa è riservata esclusivamente al servizio ACMEskyWeb, che dovrà presentare un bearer token valido rilasciato da ACMEskyService con una richiesta alla risorsa di autenticazione. Il servizio web di ACMEsky chiama questa risorsa senza parametri per recuperare la lista di tutte le offerte di interesse dell'utente che si è autenticato su ACMEskyWeb.
GET/ interests/{id}	Questa risorsa è riservata esclusivamente al servizio ACMEskyWeb, che dovrà presentare un bearer token valido rilasciato da ACMEskyService con una richiesta alla risorsa di autenticazione. ACMEskyWeb chiama questa risorsa per recuperare l'offerta di interesse corrispondente all'identificativo passato come parametro del path della richiesta.
DELETE/ interests/{id}	Questa risorsa è riservata esclusivamente al servizio ACMEskyWeb, che dovrà presentare un bearer token valido rilasciato da ACMEskyService con una richiesta alla risorsa di autenticazione. ACMEskyWeb chiama questa risorsa per cancellare l'offerta di interesse con lo stesso identificativo di quello passato come parametro del path della richiesta.
PUT/ offers/confirm	Questa risorsa è riservata esclusivamente al servizio ACMEskyWeb, che dovrà presentare un bearer token valido rilasciato da ACMEskyService con una richiesta alla risorsa di autenticazione. Il servizio web di ACMEsky chiama questa risorsa con un oggetto DTO dell'offerta di volo di interesse dell'utente come parametro per informare ACMEskyService del fatto che l'utente ha confermato l'offerta proposta inserendone il token.
PUT/ offers/paymentLink	Questa risorsa è riservata esclusivamente al servizio ACMEskyWeb, che dovrà presentare un bearer token valido rilasciato da ACMEskyService con una richiesta alla risorsa di autenticazione. Il servizio web di ACMEsky chiama questa risorsa passando il DTO dell'indirizzo dell'utente come parametro per informare ACMEskyService del fatto che lo stesso vuole acquistare l'offerta di volo per iniziare la procedura di pagamento.
PUT/ offers/reset	Questa risorsa è riservata esclusivamente al servizio ACMEskyWeb, che dovrà presentare un bearer token valido rilasciato da ACMEskyService con una richiesta alla risorsa di autenticazione. Il servizio web di ACMEsky chiama questa risorsa con un oggetto DTO dell'offerta di volo di interesse dell'utente come parametro per informare ACMEskyService di eventuali errori nella fase di

	pagamento che portano al reset del processo di conferma e acquisto dell'offerta.
GET/ offers/	Questa risorsa è riservata esclusivamente al servizio ACMEskyWeb, che dovrà presentare un bearer token valido rilasciato da ACMEskyService con una richiesta alla risorsa di autenticazione. Il servizio web di ACMEsky chiama questa risorsa per recuperare le offerte generate da ACMEsky sulla base delle preferenze dell'utente, filtrando le offerte che non sono state acquistate.
GET/ offers/{token}	Questa risorsa è riservata esclusivamente al servizio ACMEskyWeb, che dovrà presentare un bearer token valido rilasciato da ACMEskyService con una richiesta alla risorsa di autenticazione. Il servizio web di ACMEsky chiama questa risorsa passando come parametro il token presente nel path della route per recuperare l'offerta generata con quel token se è già stata acquistata, si ottiene errore.
GET/ offers/{token}/ticket	Questa risorsa è riservata esclusivamente al servizio ACMEskyWeb, che dovrà presentare un bearer token valido rilasciato da ACMEskyService con una richiesta alla risorsa di autenticazione. Il servizio web di ACMEsky chiama questa risorsa passando come parametro il token presente nel path della route per recuperare il biglietto dell'offerta se acquistata, altrimenti si riceve errore.
GET/ users/me	Questa risorsa è riservata esclusivamente al servizio ACMEskyWeb, che dovrà presentare un bearer token valido rilasciato da ACMEskyService con una richiesta alla risorsa di autenticazione. Il servizio web di ACMEsky chiama questa risorsa per recuperare le informazioni principali dell'utente che si è autenticato.
POST/ users/	Tutti i nuovi servizi e utenti possono effettuare chiamate a questa route per iscriversi nel DB di ACMEsky e interagire con i vari servizi al fine di raggiungere i propri scopi. Il servizio chiamante può effettuare chiamate a questa route passando come argomento un oggetto di tipo UserSignUpdDTO, contenente email, password, name, surname e prontogramUsername, per registrare un utente.
PUT/ users/me	Questa risorsa è riservata esclusivamente al servizio ACMEskyWeb, che dovrà presentare un bearer token valido rilasciato da ACMEskyService con una richiesta alla risorsa di autenticazione. Il servizio web di ACMEsky chiama questa risorsa passando un parametro di tipo UserUpdateDTO contenente tutti i campi necessari ad ACMEskyService per modificare la password, il nome ed il cognome.
DELETE/ users/me	Questa risorsa è riservata esclusivamente al servizio ACMEskyWeb, che dovrà presentare un bearer token valido rilasciato da ACMEskyService con una richiesta alla risorsa di autenticazione. Il servizio web di ACMEsky chiama questa risorsa passando un parametro di tipo UserDeleteDTO contenente tutti

	i campi necessari ad ACMEskyService (username e password corretti) per cancellare l'utente con codeste credenziali da ACMEsky.

ACMEskyDB

Documentazione

ACMEskyDB è l'utility che ACMEsky utilizza per creare, salvare e gestire le informazioni all'interno del DB di PostgreSQL. Il DBMS del database è pgAdmin4, dotato anche di interfaccia grafica per fare query utile nella fase di sviluppo per il testing, disponibile anche per controllare i dati salvati.

Tecnologie e implementazione

ACMEskyDB consiste in due container Docker, uno che contiene effettivamente il Database PostgreSQL e l'altro che usa un'istanza di pgAdmin4. La cartella contiene un Dockerfile che dichiara come immagine del primo container PostgreSQL versione 13.3, con l'aggiunta di un file init.sql, mentre l'immagine del secondo viene stabilita direttamente nel docker-compose file. Nel dettaglio il file init.sql contiene lo schema del DB, le configurazioni necessarie per creare le chiavi primarie ed esterne delle tabelle, alcuni record delle tabelle necessari per le interazioni di ACMEsky e degli altri servizi, come le tuple che descrivono aereoporti, compagnie aeree, banche e le rispettive entità del dominio. Le entità del dominio sono record che rappresentano i ruoli e le credenziali con cui i servizi richiedono di essere riconosciuti in quanto tali per mandare richieste al servizio di ACMEsky.

DB schema: struttura e spiegazione

	airports
PK	id
	code
	name
	city_name
	country_code
	timezone
	latitude
	longitude

La tabella "airports" contiene le tuple che rappresentano gli aereoporti nazionali e internazionali codificati secondo il codice IATA. Essa contiene il campo "id" (Chiave primaria), "code", ossia il codice dell'aereoporto in codifica IATA, "name", il nome dell'aereoporto, "city_name", il nome della città dove si trova, "country_code", iniziali del paese dove si trova, "timezone", ossia il fuso orario, "latitude" e "longitude" che rappresentano la posizione dell'aereoporto.

	domain_entities
PK	id
	username
	password
	salt
	role

La tabella "domain_entities" contiene i record che descrivono le entità del dominio, ossia gli attori che interagiscono con ACMEsky al fine di raggiungere i propri scopi. Così è possibile riconoscere il ruolo di ciascun servizio/utente in base alle proprie credenziali, evitando relazioni con parti sconosciute. Ogni tupla contiene il valore corrispondente al campo "id" (Chiave primaria), ai campi "username" e "password" (le credenziali), "salt" rappresenta un dato random addizionato all'input della funzione one-way (password) in modo da proteggere il DB da attacchi per violare password, ed al campo "role", il ruolo dell'entità.

	users
PK	id
FK	entity_id
	name
	surname
	email
	prontogram_username

La tabella "users" rappresenta gli utenti che interagiscono con il sistema. Il campo "id" è l'identificatore dell'utente nella tabella (Chiave primaria), "entity_id" è l'id con il quale è stato salvato l'utente sulla tabella "domain_entities" (Chiave esterna), "name" e "surname" sono il nome e cognome dell'utente, "email" è il campo contenente l'email con la quale l'utente si è registrato, mentre "prontogram_username" è il nome utente con il quale l'utente si è registrato sull'app di Prontogram.

	flights_interest
PK	id

FK	user_id
FK	departure_airport_id
FK	arrival_airport_id
	departure_date_time
	used

La relazione "flights_interest" descrive un volo di interesse, ossia un volo che un utente richiede attraverso il servizio di ACMEskyWeb per acquistarlo. La relazione ha un campo "id" che rappresenta il codice identificativo con il quale viene salvato sul DB (Chiave primaria), "user_id" si riferisce al campo "id" della relazione "users" (Chiave esterna), "departure_airport_id" e "arrival_airport_id" indicano l'identificatore dell'aereoporto di partenza e arrivo, "departure_date_time" descrive la data di partenza senza specificare l'ora e il campo used assume valori booleani: se assume il valore "true" indica che ACMEsky ha già proposto all'utente il volo corrispondente all'interesse.

	users_interests
PK	id
FK	user_id
FK	outbound_flight_interest_id
FK	flight_back_interest_id
	price_limit
	expire_date
	used

La tabella "users_interest" descrive le offerte di volo di interesse degli utenti. Visto che gli utenti devono sempre prenotare un volo di interesse di andata e uno di ritorno entro un certo limite di prezzo, si è deciso di rappresentare questa scelta progettuale con il nome di offerta di volo. Il campo "id" è l'identificatore dell'offerta di interesse nella tabella (Chiave primaria), "user_id" è l'identificativo dell'utente nella tabella "users" (Chiave esterna), "outbound_flight_interest_id" è l'identificativo del volo di andata nella tabella "flights_interest" (Chiave esterna), "flight_back_interest_id" è l'identificativo del volo di ritorno nella tabella "flights_interest" (Chiave esterna), il campo "price_limit" si riferisce al limite di prezzo che l'offerta non può superare, "expire_date" sta per la data di scadenza entro cui l'offerta è prenotabile, mentre "used" rappresenta con valore booleano se l'offerta di interesse è stata già gestita da ACMEsky.

	airlines
PK	id
FK	entity_id

	ws_address

La tabella "airlines" fa riferimento ai servizi delle compagnie aeree (AirlineService). Il campo "id" è l'identificativo della compagnia nella relazione (Chiave primaria), "entity_id" si riferisce all'identificativo della compagnia nella tabella "domain_entities" (Chiave esterna), e il campo "ws_address" rappresenta l'indirizzo del servizio con cui si possono fare richieste attraverso chiamate alle varie route messe a disposizione dal servizio stesso.

	flights
PK	id
FK	departure_airport_id
FK	arrival_airport_id
FK	airline_id
	flight_code
	departure_date_time
	arrival_date_time
	price
	expire_date
	booked
	available

La relazione "flights" descrive i voli che vengono recuperati interrogando la compagnia aerea sulla base dei voli di interesse degli utenti. Il campo "id" è l'identificativo del volo nella tabella (Chiave primaria), "departure_airport_id" è l'identificativo dell'aereoporto di partenza del volo con il quale è registrato nella tabella "airports" (chiave esterna), "arrival_airport_id" è l'identificativo dell'aereoporto del volo di ritorno con il quale è registrato nella tabella "airports" (chiave esterna), il campo "airline_id" è l'identificativo con il quale la compagnia aerea viene registrata nella tabella "airlines", "flight_code" è il codice con il quale il volo viene registrato dalla compagnia, "departure_date_time" sta per la data e l'orario di partenza del volo, "arrival_date_time" sta per la data e l'orario di arrivo, "price" è il prezzo, "expire_date" rappresenta la data di scadenza del volo, ossia quando non è più prenotabile, "booked" è il flag utilizzato per indicare se il volo è stato prenotato o meno, invece il flage del campo "available" stabilisce se il volo è già stato inserito in un offerta (è quindi inutilizzabile), oppure no (quindi disponibile).

	generated_offers
PK	id

FK	user_id
FK	outbound_flight_id
FK	flight_back_id
	expire_date
	total_price
	booked
	token

La relazione "generated_offers" rappresenta le offerte di volo generabili da ACMEsky sulla base delle offerte di volo di interesse degli utenti. Il campo "id" indica l'identificativo dell'offerta (chiave primaria), "user_id" è l'identificativo dell'utente nella tabella "users" (chiave esterna), "outbound_flight_id" è l'identificativo del volo di andata nella tabella "flights" (Chiave esterna), "flight_back_interest_id" è l'identificativo del volo di ritorno nella tabella "flights" (Chiave esterna), il campo "total_price" si riferisce al prezzo dell'offerta, il campo "expire_date" rappresenta la data di scadenza dell'offerta, ossia quando non è più prenotabile, "booked" è il flag utilizzato per indicare se l'offerta è stata prenotata o meno, "token" è il campo che si riferisce al token che l'utente utilizza per riscattare l'offerta.

	banks
PK	id
FK	entity_id
	ws_address

La relazione "banks" fa riferimento ai servizi bancari. Il campo "id" è l'identificativo della banca nella tabella (Chiave primaria), "entity_id" si riferisce all'identificativo della banca nella tabella "domain_entities" (Chiave esterna), e il campo "ws_address" rappresenta l'indirizzo del servizio con cui si possono fare richieste attraverso chiamate alle varie route messe a disposizione dal servizio stesso.

	rent_services
PK	id
FK	entity_id
	address
	ws_address

La tabella "rent_services" fa riferimento ai servizi di noleggio per accompagnare l'utente, eventualmente, all'aereoporto. Il campo "id" è l'identificativo del servizio nella relazione (Chiave primaria), "entity_id" si riferisce all'identificativo del noleggio nella tabella "domain_entities" (Chiave esterna), e il campo

"ws_address" rappresenta l'indirizzo del servizio con cui si possono fare richieste attraverso chiamate alle varie route messe a disposizione dal servizio stesso.

ACMEsky Web

ACMEsky Web è una single webpage application che viene utilizzata dall'utente per interfacciarsi ai servizi di ACMEsky. Non aggiunge nessuna funzionalità al sistema, il suo unico scopo è quello di fare da wrapper grafico alle API REST di ACMEsky.

Tecnologie utilizzate

- Svelte
- Typescript

Esecuzione

Installa le dipendenze

```
npm install
```

...poi fai partire Rollup:

```
npm run dev
```

L'app viene servita all'indirizzo localhost:5000

Per creare una versione ottimizzata

```
npm run build
```

AirlineService

Airline Service è il servizio che genera e invia ad ACMEsky i voli last-minute e quei voli che hanno una corrispondenza con i voli di interesse degli utenti. I voli vengono generati non appena il servizio viene attivato mentre i voli last-minute vengono generati e inviati ogni 10 minuti ad ACMEsky.

Il servizio manda i biglietti relativi ai voli che ACMEsky decide di acquistare. In caso di mancato acquisto ACMEsky chiamerà l'opportuna risorsa per comunicare l'esito negativo del pagamento e cambiare lo stato dei voli coinvolti per renderli nuovamente disponibili.

Al momento sono attive 2 istanze di AirlineService che comunicano con ACMEsky, ovvero national_airline e international_airline. La prima istanza offre voli da e verso aereoporti nazionali, mentre la seconda offre voli da aereoporti nazionali o internazionali verso quelli internazionali. Per il resto i due servizi si comportano allo stesso modo (generazione delle offerte di volo, creazione e invio automatico nel caso siano offerte last-minute, gestione dei voli acquistati dagli utenti, invio dei biglietti, ecc.). Le due istanze vengono create grazie a Docker.

Panoramica

Informazioni generali su tecnologie e scelte progettuali del servizio

Il servizio è stato realizzato con il framework Spring boot che fornisce fin da subito un minimo di configurazione per sviluppare applicazioni web JAVA e in particolare servizi REST. Si è scelto di utilizzare Apache Maven per la gestione delle dipendenze del progetto, l'importazione di tutte le librerie necessarie per sviluppo del progetto e i relativi JAR, l'esecuzione dei vari goal per fare la build e l'esecuzione del servizio. Si è deciso di usare Docker per creare due istanze di AirlineService specificando nel dockerFile la versione di Java e i vari jar da copiare ed eseguire, mentre nel docker-compose.yml si è definita la lista delle immagini dei container da creare, le variabili di ambiente racchiuse nel file di configurazione di ciascun immagine, i volumi tra cui figurano i file JSON necessari per generare le offerte di volo e il db, le porte con cui si interfacciano con l'esterno e la rete a cui sono collegati tutti i container che vengono creati.

I file JSON dei voli sono suddivisi in due liste principali: "OFFERS", la quale ospita le liste che contengono gli oggetti JSON dei biglietti per i voli "normali", mentre "LAST-MINUTE" racchiude le liste che includono al proprio interno i biglietti per i voli last-minute. Infatti ci sono due liste fatte apposta per separare i voli standard dai voli last-minute: ciascuna di queste liste contiene a sua volta un insieme di array JSON che rappresentano i voli che il servizio organizza, e ciascun array contiene gli oggetti JSON rappresentanti i biglietti disponibili per un dato volo. La creazione dei biglietti dei voli standard avviene convertendo gli oggetti JSON contenuti nelle liste in offerte di volo in seguito all'avvio del servizio, mentre la generazione dei biglietti per i voli last-minute viene fatta scegliendo casualmente tra le liste di voli la lista che contiene i biglietti da creare, che verranno a loro volta convertiti in offerte di volo ogni 10 minuti e inviati ad ACMEsky. In entrambi i casi le offerte di volo create vengono salvate sul Database. Si è scelto di convertire gli oggetti JSON dei biglietti nelle offerte di volo per non mostrare informazioni non interessanti per l'utente come il flag per cambiare lo stato di acquisto dell'offerta o la data di scadenza dell'offerta stessa, utili ai servizi per gestire i voli più comodamente. Infatti le offerte di volo vengono convertite automaticamente in nuovi oggetti chiamati "Volo" prima di essere inviati ad ACMEsky. É possibile aggiungere nuovi biglietti o rimuovere quelli già presenti integrando i file con nuove liste di oggetti JSON che rispettano le caratteristiche di biglietti, ovvero una chiave con l'opportuno nome per ogni campo e un valore adatto.

Le offerte corrispondenti ai voli di interesse degli utenti vengono recuperate dal DB con una query che cerca tutte quelle offerte che hanno gli stessi aereoporti di partenza e arrivo dei voli richiesti e che partono lo stesso giorno dei voli di interesse degli utenti, a prescindere dall'ora esatta della partenza. Vengono ignorate tutte le offerte di voli last-minute (poichè ACMEsky li ha già ricevuti) e le offerte i cui biglietti sono stati già venduti. Il servizio invia ad ACMEsky solo voli non scaduti al momento della creazione nel caso dei voli, e al momento dell'invio se si tratta di voli last-minute. Prima di inviare le offerte last-minute il servizio

chiede ad ACMEsky di autenticarsi e, in caso di esito positivo, riceve un token jwt da passare nel campo "authorization" dell'header della richiesta di invio delle offerte.

Per la generazione dei biglietti aerei da inviare all'utente si è scelto di usare le librerie di Thymeleaf, che offre la possibilità ad applicazioni server side di realizzare template in grado di eseguire codice HTML e CSS in maniera semplice, oltre al loro riempimento con le informazioni del biglietto di volo ed alla generazione di file pdf.

Il DBMS che è stato scelto è quello di H2, scritto in Java con tecnologia in-memory, tra l'altro accessibile attraverso le route e le credenziali stabilite nel file application.properties, le cui impostazioni stabiliscono che il DB mantiene il suo stato anche se il server viene spento.

Struttura del servizio

Il progetto è stato strutturato sulla base della Dependency Injection di Spring boot, quindi vi è il package service contenente i servizi ("FlightOfferService" e "PdfService") che vengono "iniettati" nel "FlightOfferController" presente nel package controller per offrire le feature richiamabili dalle risorse che verranno descritte in seguito. Il "FlightOfferService" si occupa di fornire i metodi e le funzioni richiamabili dalle risorse del controller per garantirne il funzionamento. Vi sono i metodi e le funzioni per la creazione, ed il salvataggio delle offerte di volo nel Database e per l'eventuale invio dei voli nel caso siano last-minute, le funzioni per cercare le offerte di volo compatibili con i voli di interesse degli utenti e l'invio dei rispettivi voli, il metodo per il cambiamento di stato delle offerte di volo da disponibili ad acquistate ed eventualmente il viceversa. Il "PdfService" si occupa di creare file pdf contenenti tutte le informazioni sui biglietti che l'utente vuole acquistare. Il package model include la classe per la definizione dell'offerta di volo, ovvero "FlightOffer", la classe per la definizione di alcune utility per la generazione e gestione delle offerte, ovvero "FlightUtility". E'stato creato il package per i Data Transfer Object (DTO) per rappresentare le richieste degli utenti ("UserRequest"), per rappresentare i voli che trovano una corrispondenza con gli interessi degli utenti ("Flight"), e il DTO per la richiesta di autenticazione chiamato "AuthRequest", impiegato per richiedere un token jwt per autenticarsi nel caso i servizi lo richiedono per fare richieste HTTP, come ad esempio ACMEsky. Inoltre è presente un package repository che contiene un interfaccia che estende una repository JPA utilizzata che consente di interfacciarsi con la tabella contenente le offerte di volo per la gestione delle funzionalità che offre il servizio di Airline. Nella directory resources vi sono le risorse necessarie per realizzare i biglietti aerei, ovvero il template "Ticket_template" e la sottodirectory "pdf-resources" che ospita la cartella css contentente un css impiegato dal template.

URI

Tutti gli URI riferiti ai vari container che ospitano i servizi di AirlineService sono i seguenti:

- http://localhost:8060 per airlineservice_national
- http://localhost:8061 per airlineservice_international

API:

```
http://localhost:8060/swagger-ui.html
http://localhost:8061/swagger-ui.html
```

Risorse e descrizione

Risorsa	Descrizione
POST /getFlights	Questa risorsa consente di richiedere i voli che corrispondono ai voli di richiesta degli utenti passati come parametro. Infatti prende in input una lista di oggetti JSON che vengono deserializzati grazie a Jackson Json in oggetti di tipo UserRequest e cerca nella repository le offerte che hanno gli stessi aereoporti di partenza e arrivo e la stessa data di partenza (non si considera l'orario, infatti si cerca tutti i voli disponibili per l'intero giorno di andata). Se queste non sono offerte last-minute e non sono già state acquistate da altri utenti, vengono convertite in oggetti Flight e poi inviate in risposta alla chiamata, altrimenti no.
POST /notPurchasedOffer	Le chiamate a questa risorsa consentono di cambiare lo stato delle offerte di volo che non vengono acquistate in seguito ad eventuali errori da parte di ACMEsky, in modo da renderle nuovamente disponibili.
GET /getTickets	Le chiamate a questa risorsa che hanno come parametro la lista di identificatori delle offerte che l'utente ha intenzion di acquistare consentono di ricevere i biglietti dei voli in formato pdf. Nello specifico si cambia lo stato di acquisto delle offerte corrispondenti ai voli che si vuole acquistare e si restituisce un file che elenca e descrive brevemente le caratteristiche dei voli. Infine si imposta il tipo del contenuto del risultato, ovvero un pdf, e gli header che stabiliscono che vi un file in allegato alla risposta.

Build fat Jar:

```
mvn package
```

Come eseguire

```
mvnw spring-boot:run
```

Build e run con Docker compose

```
docker-compose up --build
```

Credenziali database

DB console service 1

```
http://localhost:8060/h2
URL: jdbc:h2:file:./db/db
user: sa
passw:
```

DB console service 2

```
http://localhost:8061/h2
URL: jdbc:h2:file:./db/db
user: sa
passw:
```

Bank service

Bank è il servizio con cui ACMEsky si interfaccia per la gestione dei pagamenti. ACMEsky richiede a Bank i link di pagamento che poi verrano utilizzati dall'utente per pagare. Inoltre una volta effettuato il pagamento invia un messaggio ad ACMEsky con la relativa conferma. Tutte le richieste che vengono fatte alla banca devono esere autenticate utilizzando il token JWT che può essere richieste utilizzando la route /path.

API:

```
http://localhost:8070/swagger-ui.html
```

Build fat Jar:

```
mvn package
```

Come eseguire

```
mvnw spring-boot:run
```

Build e run con Docker compose

```
docker-compose up --build
```

Credenziali database

```
http://localhost:8070/h2
URL: jdbc:h2:./db/bankdb
user: sa
passw:
```

ProntoGram
Prontogram è una applicazione web che riceve le offerte dei voli da ACMEsky, attraverso un sistema di notifiche web. Le offerte ricevute vengono generate da ACMEsky, dopo che l'utente ha inserito su ACMEsky le preferenze su dei voli di proprio interesse. L'offerta comprende sia il volo di andata si il volo di ritorno. L'istanza dell'offerta ricevuta è in formato JSON, più precisamente è un array che contiene l'username dell'utente al quale è indirizzata la notifica dell'offerta e un messaggio scritto in HTML, il quale verrà utilizzato per rappresentare graficamente il messaggio all'interno della sezione notifiche di Prontogram. L'applicazione web di Prontogram si divide in due parti: front-end e back-end. Entrambe vengono istanziate grazie a Docker, in particolare al docker-compose. La parte back-end rappresenta il server di Prontogram e si occupa di gestire tutte le chiamate delle API da e verso la parte client di Prontogram. Mentre la parte fornt-end si occupa di creare l'interfaccia grafica e gestire le iterazioni da parte dell'utente.

Informazioni generali su tecnologie e scelte progettuali del servizio

Come precedentemente enunciato, Prontogram si divide in due parti rispettivamente, parte front-end e parte back-end. Ognuna di queste è stata implementata utilizzando tecnologie e pattern strutturali diversi.

Front-end

La parte front-end è stata realizzata con il framework Angular, il quale consente di scomporre l'interfaccia utente in blocchi gestibili e di separare l'interfaccia utente dall'implementazione rendendo la generazione di pagine lato server molto più semplice. L'architettura modulare di Angular consente di strutturare al meglio un'applicazione e permette di semplificare il processo di creazione di SPA (Single page application). La parte front-end di Prontogram è stata implementata in tre macro componenti:

- AccountComponent : componente che gestisce la parte di login e registrazione dell'utente;
- NotificationComponent : componente che gestisce le notifiche delle offerte ricevute da ACMEsky;
- UserComponent : componente che gestisce le informazioni relative all'utente, permettendo di modificare le informazioni inserite da quest'ultimo durante la fase di registrazione alla web application. Per quanto riguarda le tecnologie usate per implementare il sistema di notifiche, è stato utilizzato il servizio "service worker" fornito da Angular. La comunicazione tra applicazione web e server viene stabilita utilizzando una coppia di chiavi VAPID. VAPID è l'acronimo di "Voluntary Application Server Identification" per il protocollo Web Push. Una coppia di chiavi VAPID è una coppia di chiavi crittografiche pubbliche/private che viene utilizzata nel seguente modo:
- Chiave pubblica: viene utilizzata come identificatore univoco del server per iscrivere l'utente alle notifiche inviate da quest'ultimo;
- Chiave privata : deve essere tenuta segreta (a differenza della chiave pubblica) e viene utilizzata dall'application server per firmare i messaggi, prima di inviarli al servizio Push per la consegna del messaggio.

Back-end

La parte back-end di Prontogram è stata realizzata utilizzando Node.js, un runtime system open source multipiattaforma orientato agli eventi per l'esecuzione di codice JavaScript. Più nello specifico è stato utilizzato Express.js un web framework per Node.js, il quale offre strumenti di base per creare più velocemente applicazioni in Node. Express.js ha permesso di realizzare un server abbastanza facilmente andando a generare le route (URL) utilizzate da ACMEsky e Prontogram durante le chiamate REST. Il package models include la classe per la definizione della notifica di un'offerta, ovvero notification, la classe per la definizione dell'utente, ovvero user, e la classe per la definizione della sottoscrizione al server, ovvero subscription. Inoltre è presente un package routes il quale contiene tutti i path/percorsi per gestire le chiamate alle API del server. Per salvare i dati ricevuti e inviati dal server è stato utilizzato MongoDB un DBMS non relazionale, orientato ai documenti.

API

API link

Risorse e descrizione

Risorsa	Descrizione
POST/ notification/posts/	Esegue la creazione della notifica e il salvataggio della notifica nel database poi inviarla a Prontogram attraverso il metodo sendNotification(). L'invio dell'offerta avviene in seguito ad una ricerca nel database dell'utente al quale è indirizzata. Gli utenti prima di ricevere le notifiche delle offerte dei voli devono aver eseguito la sottoscrizione al server, la quale avviene in automatico dopo aver effettuato il login all'applicazione web Prontogram.
DELETE/ notification/posts/:notificationId	Esegue la cancellazione di una notifica di un'offerta attraverso il passaggio come parametro dell'Id della notifica.
GET/ notification/gets/all/:username	Esegue il caricamento di tutte le notifiche delle offerte, attraverso il passaggio come parametro dell'username dell'utente corrispondente.
POST/ subscription/post/new	Esegue la creazione e il salvataggio in database della sottoscrizione di un'utente al servizio di notifiche. Quest'ultima avviene in automatico dopo aver effettuato il login all'applicazione web Prontogram.
DELETE/ subscription/posts/:subendpoint	Esegue l'annullamento della sottoscrizione e la cancellazione di quest'ultima dal database. La chiamata a questa route avviene in automatico dopo aver effettuato il logout dall'applicazione web Prontogram.
GET/ subscription/gets/all	La chiamata a questa route permette di reperire tutte le sottoscrizioni effettuate dagli utente al server.

POST/ user/posts/	Esegue la creazione di un nuovo utente e il suo salvataggio nel database.
DELETE/ user/posts/:userId	Esegue l'eliminazione di un utente dal database, attraverso il passaggio come parametro dell'Id dell'utente corrispondente.
PUT/ user/posts/:userId	Esegue l'update delle informazioni relative ad un'utente, attraverso il passaggio come parametro dell'Id dell'utente corrispondente.
GET/ user/gets/:id	La chiamata a questa route permette di reperire le informazioni relative ad un'utente, attraverso il passaggio come parametro dell'Id dell'utente corrispondente.
GET/ user/gets/all	La chiamata a questa route permette di reperire tutte le informazioni relative a tutti gli utenti presenti in database.
POST/auth/register	Esegue la registrazione di un utente a Prontogram andando a creare e salvare in database tutte le informazioni relative ad un'utente inserite durante la fase di registrazione.
POST/auth/login	Esegue il login da parte di un utente a Prontogram.

URI

Tutti gli URI riferiti ai vari container che ospitano i servizi di Prontogram sono i seguenti:

```
- http://localhost:8050 per Prontogram back-end
- http://localhost:8051 per Prontogram front-end
- http://localhost:8052 per MongoDB
```

How to build

```
cd front-end
npm install
ng build --prod
```

How to run

```
cd static
http-server
```

Run on Docker Compose

```
docker-compose up --build
```

Rent Service

Servizio che simula una compagnia di noleggio.

Utilizza SOAP per esporre i servizi.

Service ports

Name	Endpoint (Location)
RentServicePort	http://localhost:8080

PortType: Rent

Operation	Input	Output
bookRent	RentRequest	RentResponse

Bindings

Name	Type	PortType	Style
RentSOAPBinding	SOAP11	Rent	Document/Literal-Wrapped

Run:

```
jolie server.ol $SERVICE_NAME
```

Create and run the docker stack

```
docker-compose up
```

Geographical Distance Service

Servizio che si occupa di calcolare la distanza tra due punti specificati tramite indirizzo o coordinate geografiche. Internamente utilizza le api di distancematrix.ai per il calcolo delle distanze.

Run:

```
npm install
node index.js -p 8080
```

Run con Docker compose

```
docker-compose up
```

API:

Metodo	Path	Parametri
GET	/distance	from: Luogo di partenza; to:Luogo di arrivo.

Esempio

```
http://localhost:8080/distance?from=Mura+Anteo+Zamboni+7+40126+Bologna+%28BO%29%0D%0A&to=Ferrara
```