

Università di Bologna

Progetto di Ingegneria del Software Orientata ai servizi
ACMEsky

Baratin Riccardo
Di Ubaldo Andrea
Vallorani Giacomo

Anno accademico 2020/2021

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 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 richiesta 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 quella 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 aggiuntivi

Durante il design del sistema sono stati aggiunti dei vincoli per raffinare le specifiche del progetto.

- Ogni attore deve essere autenticato e autorizzato per poter interagire con ACMESky;
- I voli delle offerte generate provengono dalla stessa compagnia aerea;
- La compagnia aerea restituisce una lista di voli (non necessariamente A/R);

- 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 accessori.

Servizi implementati

- **ACMEsky:** Servizio che genera offerte di voli A/R in base agli interessi degli utenti.
- **Airline Service:** Compagnia aerea.
- **Bank Service:** Gestore dei pagamenti.
- **Prontogram:** App di messaggistica.
- **GeoDistance Service:** Servizio per il calcolo delle distanze geografiche.
- **Rent Service:** Servizio di noleggio con conducente.

Struttura della documentazione

- [Coreografie](#)
- [Diagrammi BPMN](#)
- [Coreografie BPMN](#)
- [Diagrammi UML](#)
- [Struttura](#)
- [Esecuzione](#)
- Servizi
 - ACMEsky
 - [Service](#)
 - [Database](#)
 - [Web](#)
 - [Airline Service](#)
 - [Bank Service](#)
 - [Prontogram](#)
 - [Rental Service](#)
 - [GeographicalDistance Service](#)

Coreografie

In questa sezione vengono mostrate le coreografie che descrivono l'interazione tra i diversi attori.

Nomenclatura

Nome	Sigla	Commento
ACME	ACME	
Airline service	AIR _k	Indica la k-esima compagnia aerea
Rental service	RENT _t	Indica la t-esima compagnia di noleggio
Prontogram	PTG	
Bank service	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: Richiesta 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
// responseLastMinute: risposta successo o fallimento
( sendLastMinute: AIRk -> ACME ; responseLastMinute: 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: mesaaggio di offerta A/R
// notifyUser: messaggio di notifica di Prontogram
// notifyResponse: risposta da parte dell'utente dell'avvenuta ricezione
// messageSended: risposta da parte di prontogram dell'avvenuto invio
( offerToken: ACME -> PTG ; notifyUser: PTG -> USERx ;
  notifyResponse: USERx -> PTG ; messageSended: PTG -> ACME )*
|
// Richiesta ticket
// getInvoice: mesaaggio di richiesta ricevuta dell'offerta pagata
// invoice: messaggio con la ricevuta del viaggio
( getInvoice: USERx -> ACME ; invoice: ACME -> USERx )*
|
// Conferma dell'offerta e pagamento
// confirmOffer: messaggio di conferma offerta e pagamento
(
  confirmOffer: USERx -> ACME ;
  (
    // ACMEsky conferma che l'offerta è disponibile
    // responseOfferOk: messaggio di conferma offerta
    // requestPaymentLink: richiesta di pagamento da parte dell'utente
    (
      responseOfferOk: ACME -> USERx ;
      requestPaymentLink: USERx -> ACME ;
      bookTickets: ACME -> AIRk ;
      (
        // 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
        (
          responseTickets: AIRk -> ACME ;
          requestBankLink: ACME -> BANK ;
          responselink: BANK -> ACME ;
          paymentLink: ACME -> USERx ;
          payment: USERx -> 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 noleggio andata
      // requestRentReturn: richiesta noleggio ritorno
      // responseRentReturn: risposta noleggio ritorno
      requestRentDeparture: ACME -> RENTi ;
      responseRentDeparture: RENTi-> ACME ;
      requestRentReturn: ACME -> RENTi ;
      responseRentReturn: RENTi-> ACME
    )
  )
)
+
(
  // Errore nel pagamento
  // unbookTickets: cancella la prenotazione dei biglietti
  // emitCoupon: pagamento fallito
  unbookTickets: ACME -> AIRk ;
  emitCoupon: ACME -> BANK
)
)
)
)
// Errore nella prenotazione dei biglietti
// flightNotFound: volo non trovato
// errorTickets: errore volo non disponibile
+
(
  flightNotFound: AIRk -> ACME ;
  errorTickets: ACME -> USERx
)
)
)
+
// ACMEsky controlla l'offerta e non è più disponibile
// responseOfferError: errore offerta
responseOfferError: ACME -> USERx
)
)*

```

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 -> AIRk ; responseFlights: AIRk -> 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: AIRk -> ACME ; responseLastMinute: ACME -> AIRk )*
```

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

Registrazione interesse dell'utente

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

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 -> USERx ; notifyResponse: USERx -> 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: USERx -> ACME ; invoice: ACME -> USERx )*
```

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: USERx -> ACME ;
```

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

```
2.   (
3.     ( responseOfferOk: ACME -> USERx ; requestPaymentLink: USERx -> ACME ; bookTickets: ACME
-> AIRk
```

E' connessa per la sequenza in quanto il ricevente di ***responseOfferOk*** è il mittente di ***requestPaymentLink***, il ricevente di ***requestPaymentLink*** è il mittente di ***bookTickets***.

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

E' connessa per la sequenza in quanto il ricevente di ***responseTickets*** è il mittente di ***requestBankLink***, 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 -> RENTt ; responseRentDeparture: RENTt-> ACME
;
16.          requestRentReturn: ACME -> RENTt ; responseRentReturn: RENTt-> 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 -> AIRk ; unbookTicketsResponse: AIRk -> 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.          ) + ( flightNotFound: AIRk -> ACME ; errorTickets: ACME -> USERx )
```

E' connessa per la choice perché i mittenti di (6) e di (25) sono gli stessi. E' connessa per la sequenza perché il destinatario di **flightNotFound** è il mittente di **errorTickets**.

```
26.      )
27.      ) + responseOfferError: ACME -> USERx
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@AIRk ; responseFlights@AIRk )*
```

```
proj(RicezioneOfferteLastMinute, ACME) =
  _____
  ( sendLastMinute@AIRk ; repsonseLastMinute@AIRk )*
```

```
proj(RegistrazioneInteresse, ACME) =
  _____
  ( requestInterest@USERx ; responseInterest@USERx )*
```

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

```
proj(RichiestaRicevuta, ACME) =
  _____
  ( getInvoice@USERx ; invoice@USERx )*
```

```

proj(AcquistoOfferta, ACME) =
( confirmOffer@USERx ;
(
  (responseOfferOk@USERx ; requestPaymentLink@USERx ; bookTickets@AIRk
  (
    (
      responseTickets@AIRk ;

      requestBankLink@BANK ; responselink@BANK ;

      paymentLink@USERx ; 1 ;
      (
        (
          successPaymentBank@BANK ;

          ( 1 + ( requestDistance@GEO ; responseDistance@GEO ;

            ( 1 + ( ( requestDistanceRent@GEO ; responseDistanceRent@GEO )* ;

              requestRentDeparture@RENTt ; responseRentDeparture@RENTt ;

              requestRentReturn@RENTt ; responseRentReturn@RENTt
            ) )
          ) )
        ) + ( unbookTickets@AIRk ; emitCoupon@BANK )
      )
    )
  ) + flightNotFound@AIRk errorTickets@USERx
)
) + responseOfferError@USERx
)
)*

```

Utente

```

proj(QueryDeiVoli, USERx) =
( 1 ; 1 )*

```

```

proj(RicezioneOfferteLastMinute, USERx) =
( 1 ; 1 )*

```

```

proj(RegistrazioneInteresse, USERx) =
( requestInterest@ACME ; responseInterest@ACME )*

```

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

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

```
proj(AcquistoOfferta, USERx) =  
  ( 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 )  
        )  
    ) + ( 1 ; errorTickets@ACME )  
  )  
 ) + responseOfferError@ACME  
 )  
 )*
```

Airline service

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

```
proj(RicezioneOfferteLastMinute, AIRk) =  
  ( sendLastMinute@ACME ; responseLastMinute@ACME )*
```

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

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

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

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

Bank service

```
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@USERx ;
(
(
successPaymentBank@ACME ;

( 1 + ( 1 ; 1 ;
( 1 + (( 1 ; 1)* ; 1 ; 1 ; 1 ; 1 ))
))
) + ( 1 ; emitCoupon@ACME )
)
) + ( 1 ; 1 )
)
) + 1
)
)*

```

Geographical Distance 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 )
)
) + 1
)
)
)*

```

Rental Service

```

proj(QueryDeiVoli, RENTi) =
( 1 ; 1 )*

```

```

proj(RicezioneOfferteLastMinute, RENTi) =
( 1 ; 1 )*

```

```

proj(RegistrazioneInteresse, RENTi) =
( 1 ; 1 )*

```

```

proj(NotificaOfferta, RENTi) =
( 1 ; 1 ; 1 ; 1 )*

```

```

proj(RichiestaRicevuta, RENTi) =
( 1 ; 1 )*

```

```

proj(AcquistoOfferta, RENTi) =
( 1 ;
(

```

```
( 1 ; 1 ; 1 ;
(
( 1 ; 1 ; 1 ; 1 ; 1 ;
(
( 1 ;
( 1 + ( 1 ; 1 ;
( 1 + (( 1 ; 1)* ;

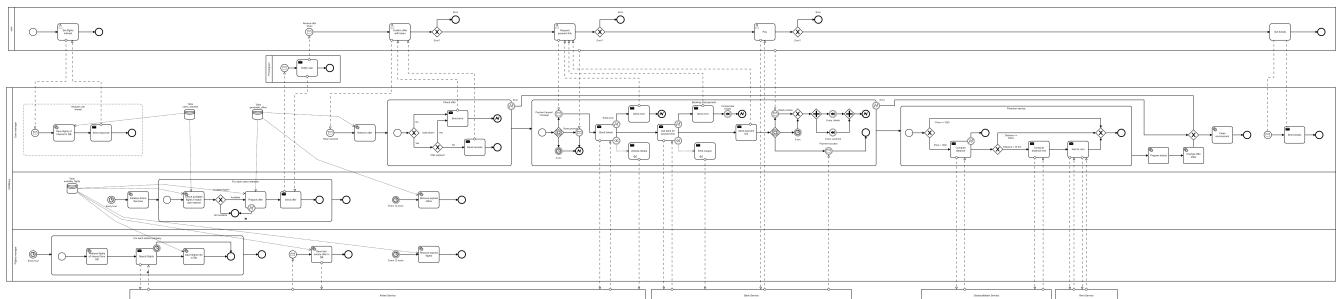
requestRentDeparture@ACME ; responseRentDeparture@ACME ;

requestRentReturn@ACME ; responseRentReturn@ACME ))
))
) + ( 1 ; 1 )
)
) + ( 1 ; 1 )
)
) + 1
)
)*
```

Diagramma BPMN

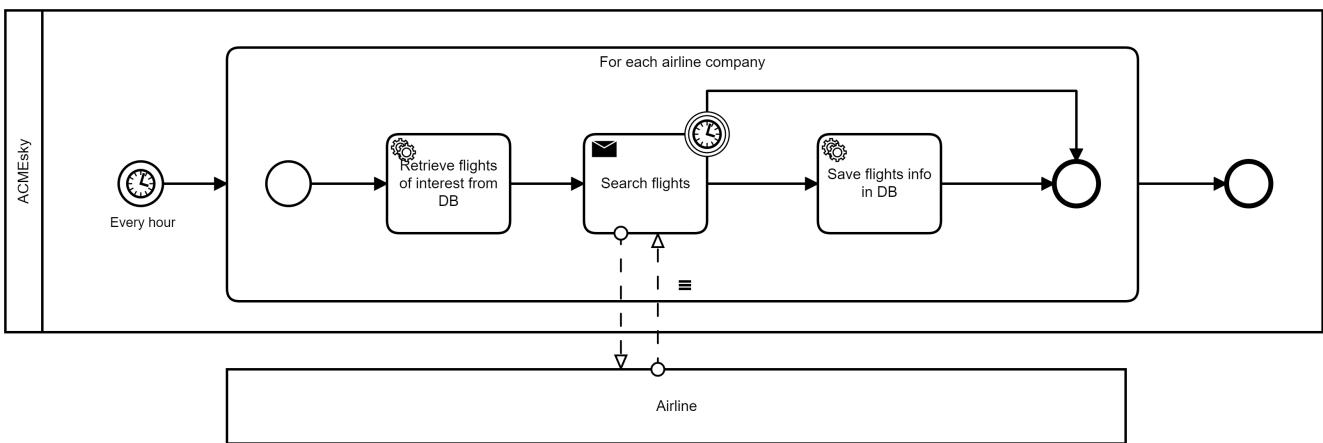
In questa sezione della documentazione viene descritto il diagramma BPMN che rappresenta il comportamento dei processi per ciò che concerne il flusso di controllo.

Diagramma completo



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 dei voli last-minute, gestione delle offerte, pagamento, ecc.

Ricerca dei voli

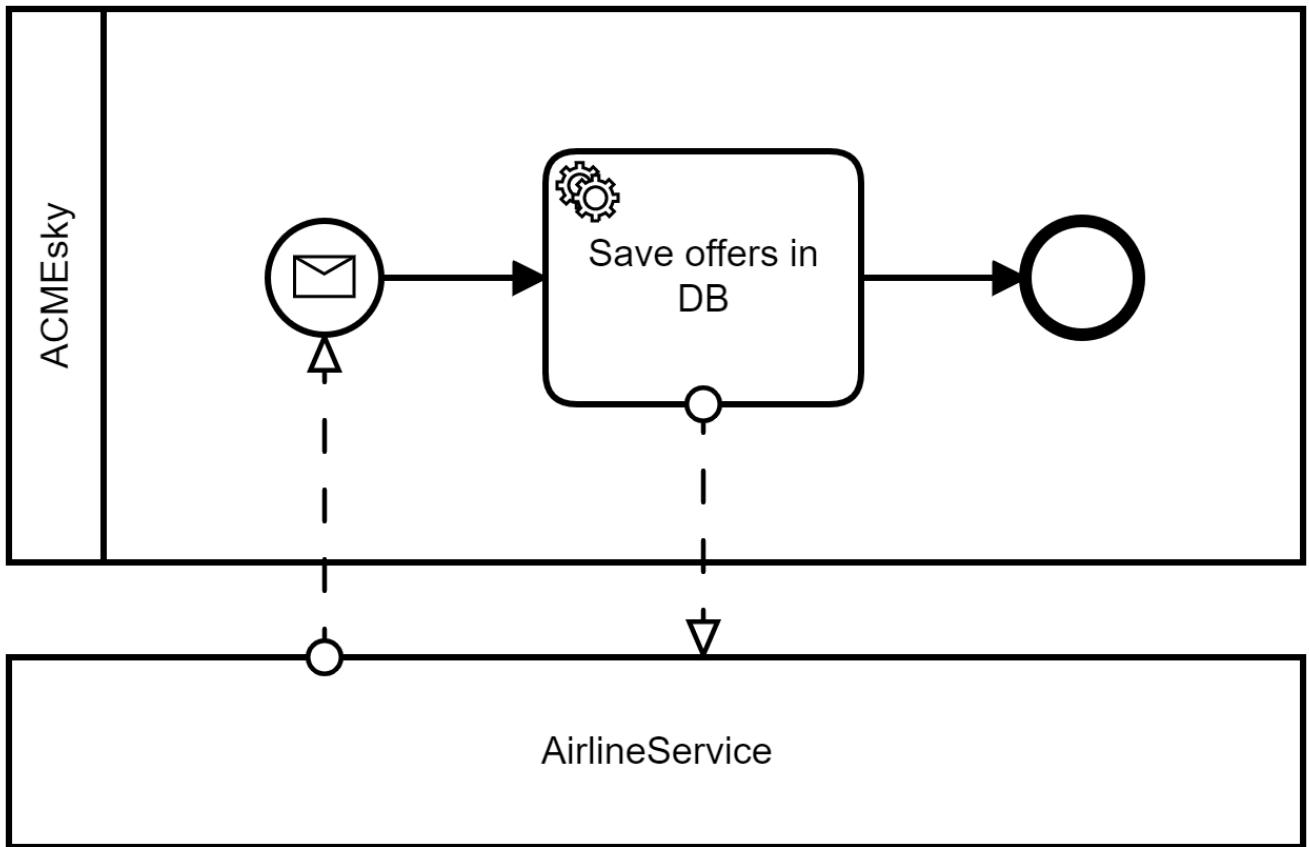


La ricerca dei voli alle Airline Services viene ripetuta ad un certo intervallo di tempo, per evitare sovraccaricare i sistemi. Per questo, i voli delle offerte di interesse degli utenti vengono salvati sul DB e, in seguito, vengono utilizzati per cercare i voli delle compagnie aeree. L'intervallo di 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. I voli presenti nella risposta

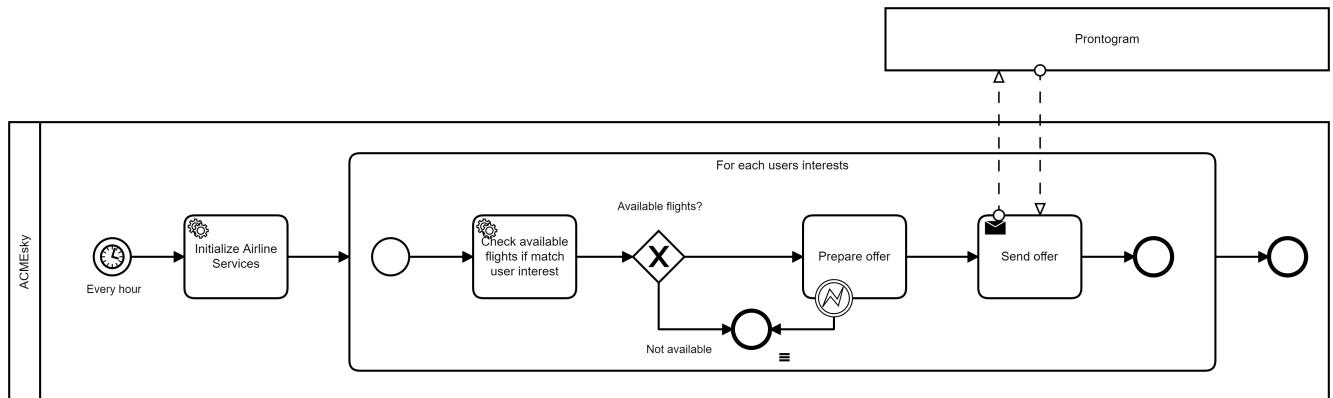
vengono salvati all'interno del database (tabella *available_flights*). Se il timer della richiesta scade, per eventuali errori dovuti ad **Airline Service**, il sottoprocesso termina e si passa ad un'altra compagnia aerea.

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 generati ad **ACMEsky**, la quale salva ciascuno di essi nel database, nello specifico nella tabella *available_flights*.

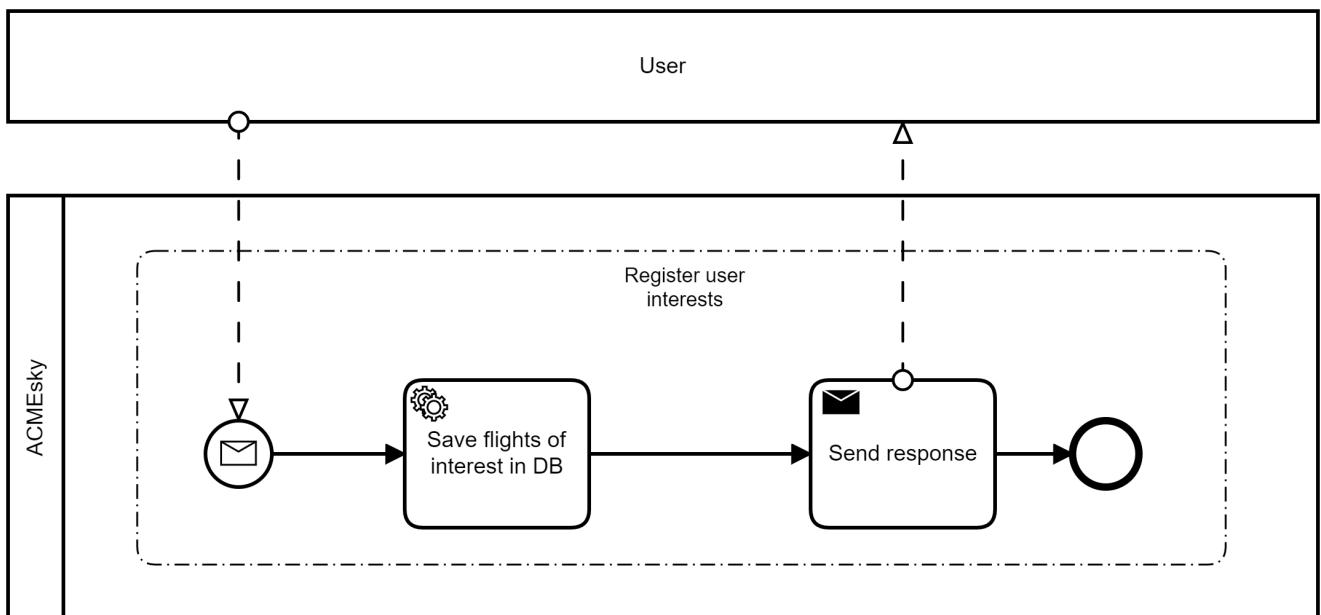
Match voli con interesse utente



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 finché non vengono processati per generare le offerte corrispondenti. L'intervallo di tempo è variabile da un'ora a pochi minuti poiché si cerca un compromesso tra un sistema efficiente e un sistema che non faccia aspettare l'utente.

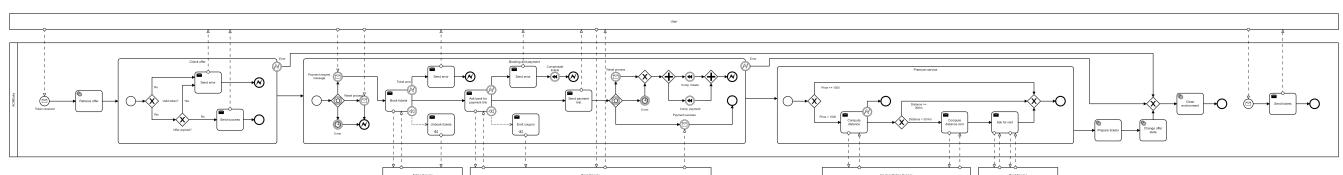
Ogni ora, per ciascun offerta di interesse, **ACMEsky** cerca tra i voli disponibili presenti nel database (tabella *available_flights*), se c'è una corrispondenza con i voli di interesse degli utenti (tabella *flights_interest*), allora prepara l'offerta, la salva nel database e la invia all'utente attraverso l'app di messaggistica **Prontogram**. In caso negativo semplicemente il sotto-processo 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/aeroporto di partenza, città/aeroporto 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*, quest'ultima contiene i voli di interesse per uno specifico viaggio. Infine, **ACMEsky** invia la conferma di avvenuta creazione.

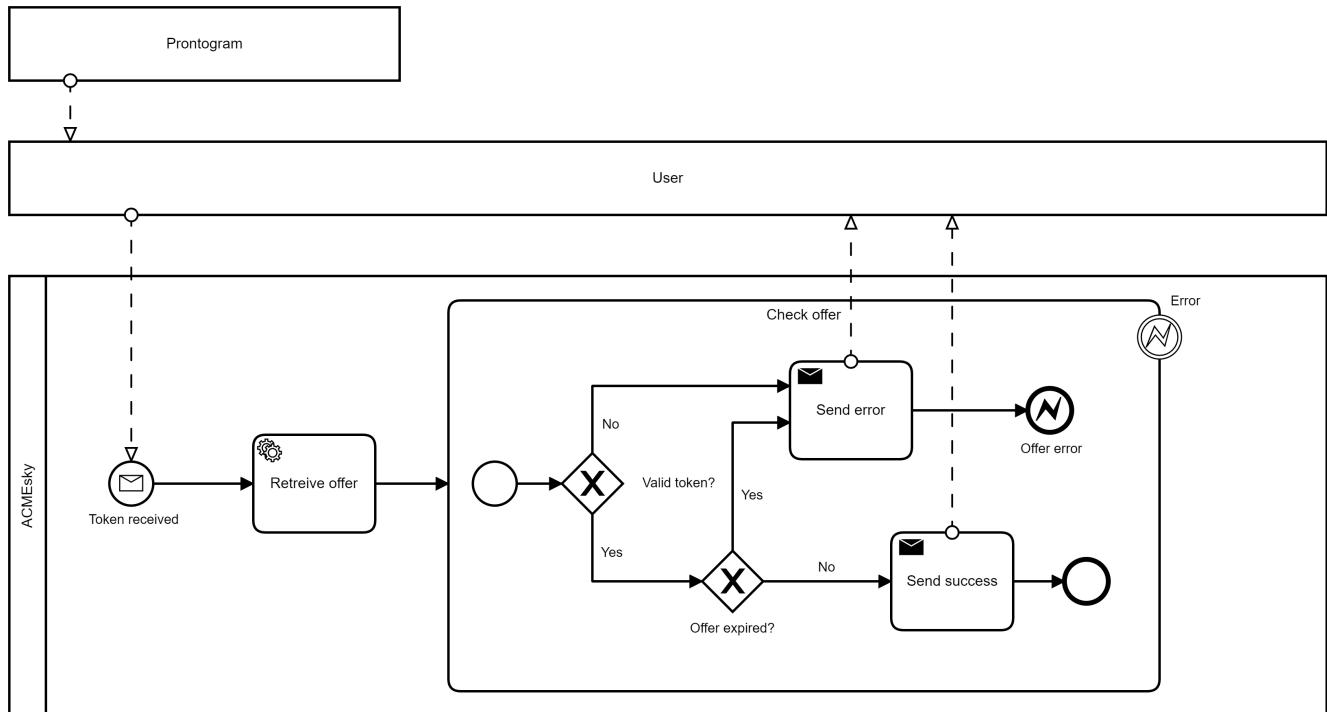
Conferma di acquisto, applicazione servizi premium e preparazione biglietti



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 maggiore comprensione il diagramma è stato diviso in blocchi più piccoli.

Conferma dell'offerta da parte dell'utente

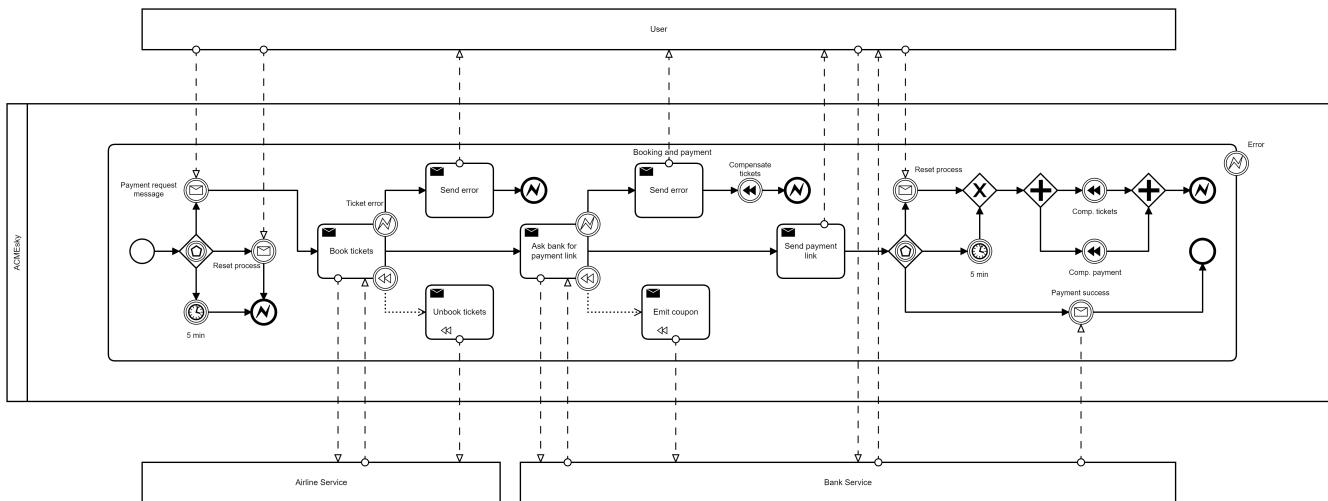


L'app di **Prontogram** notifica l'utente del fatto che c'è un'offerta disponibile.

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 non sia scaduta e anche in questo caso se 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 processo termina con un errore.

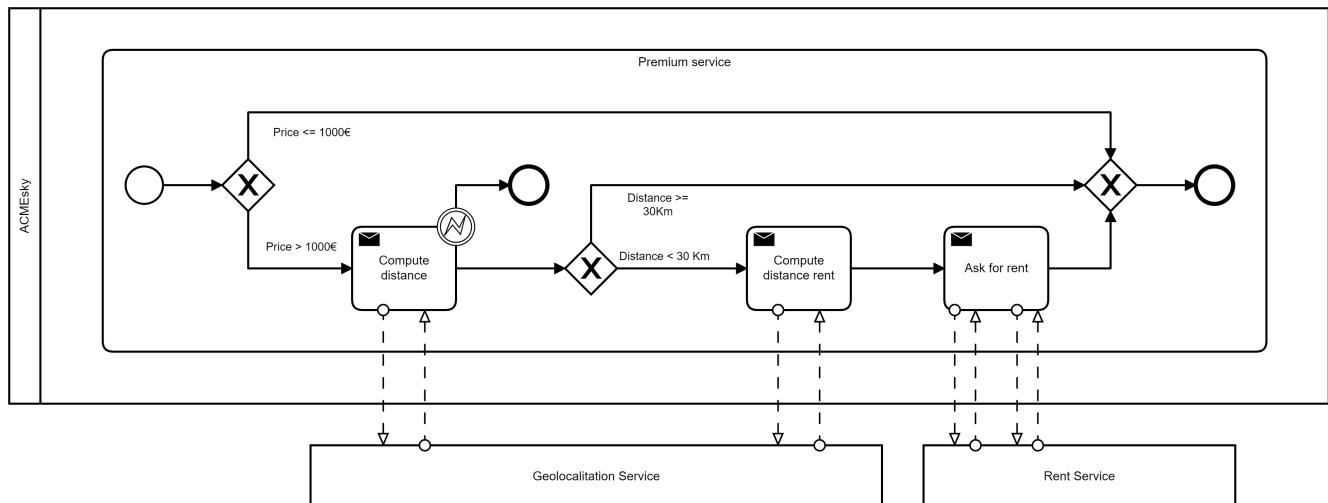
Pagamento dell'offerta



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. Se c'è un errore relativo all'impossibilità di prenotare l'offerta, poiché 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 processo 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 di pagamento. 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, in via preventiva, alla compensazione dei biglietti e del pagamento. In questo caso il processo termina con errore.

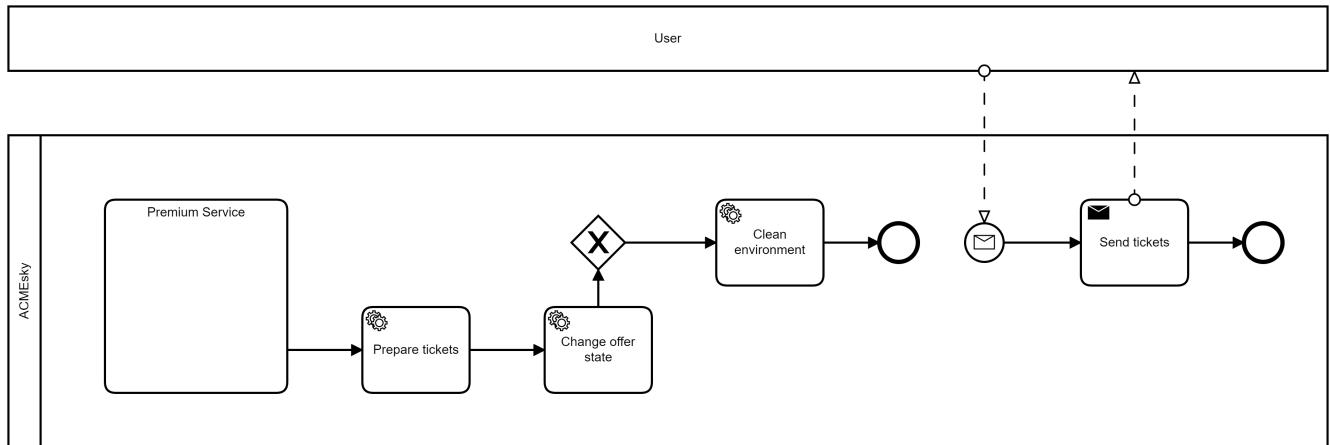
Servizi premium



In questa fase del processo di conferma, a condizioni rispettate, vengono applicati all'offerta i servizi premium. Inizialmente **ACMEsky** controlla il prezzo dell'offerta, se questo supera i 1000€ invia una richiesta al servizio di **GeoDistance** per calcolare la distanza dell'utente dall'areoporto. Nel caso in cui la distanza sia

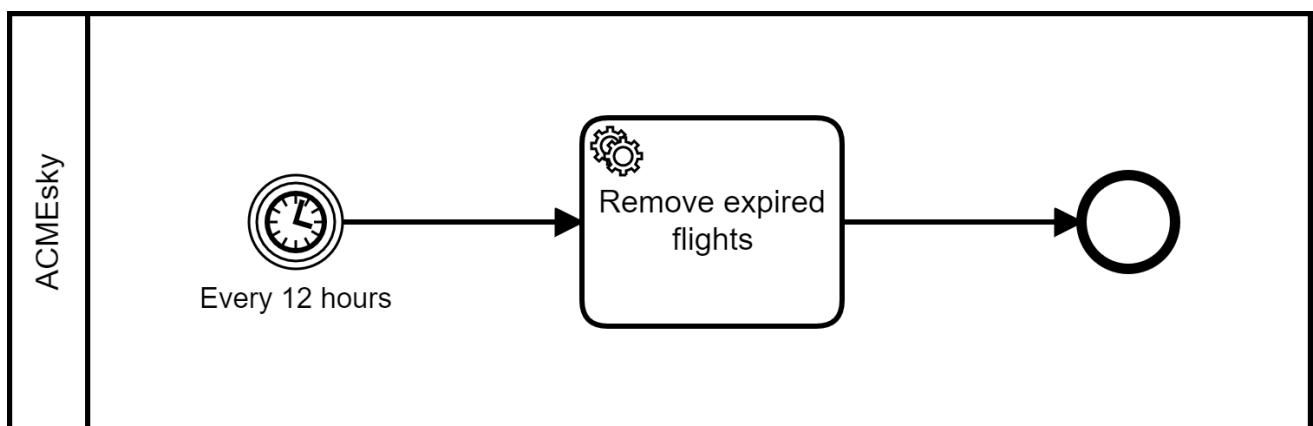
superiore ai 30 km si richiede al **Rent Service** più vicino se c'è la possibilità di offrire all'utente un trasferimento dal suo domicilio all'aeroporto. Questa operazione viene ripetuta sia all'andata che al ritorno, e in tal caso verranno modificati i biglietti includendo le informazioni dei trasferimenti.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

Preparazione dei biglietti



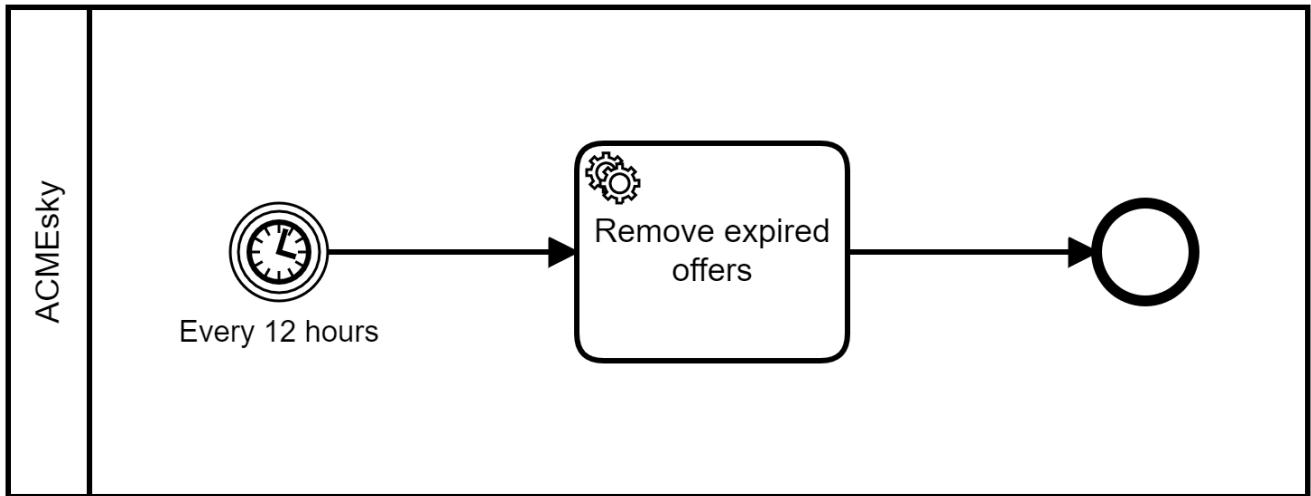
Arrivati a questo punto viene cambiato lo stato dell'offerta e viene preparato il pdf contenente i biglietti che l'utente potrà scaricare. L'utente può in qualunque momento richiedere i biglietti che ha acquistato.

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

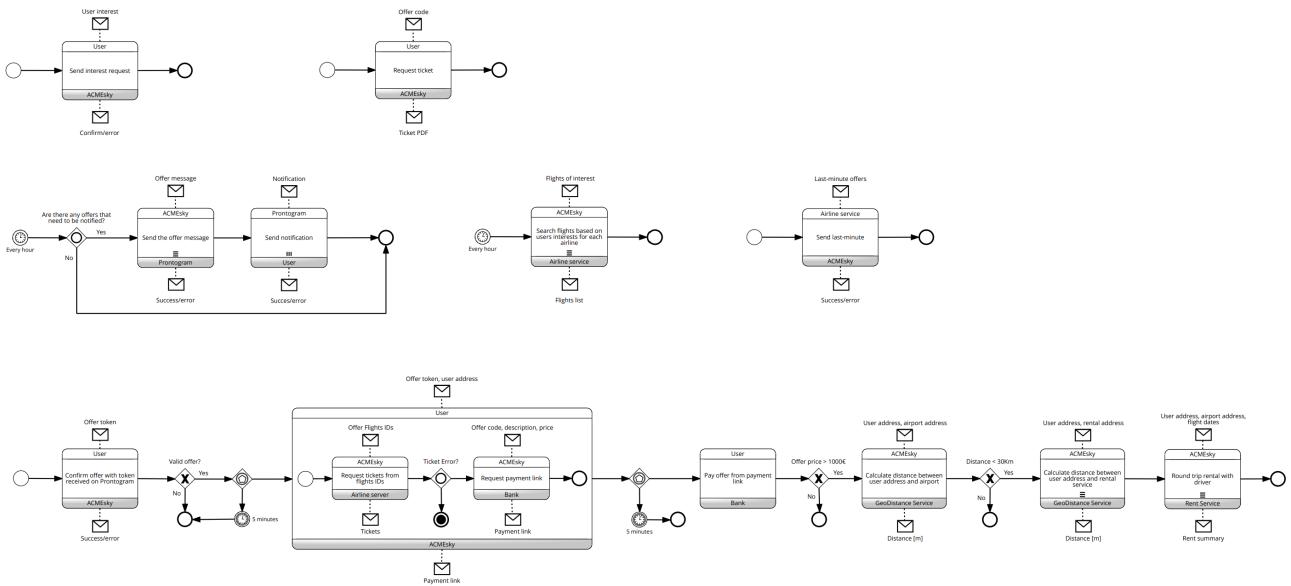


Il processo di cancellazione delle offerte scadute presenti nel Database avviene ogni 12 ore. Le offerte di volo scadute comprendono i voli di andata e ritorno. Le offerte scadute sono quelle la cui data di scadenza del volo di partenza è antecedente a quella in cui si effettua l'operazione di cancellazione. 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

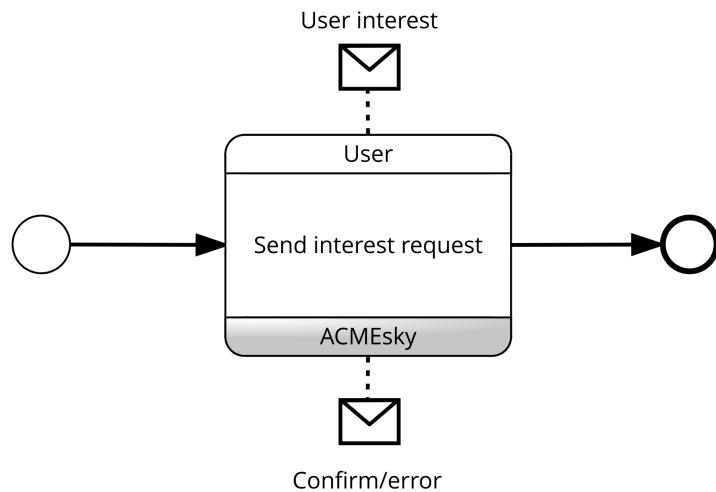
In questa sezione vengono mostrate le coreografie BPMN che mostrano le interazioni tra i processi relativi ai diversi partecipanti.

Diagramma completo



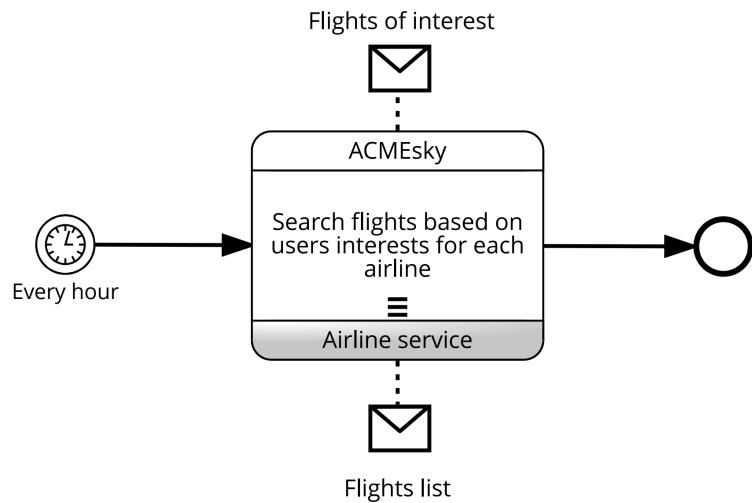
Il diagramma è stato suddiviso in parti per poterle spiegare singolarmente.

Salvataggio degli interessi



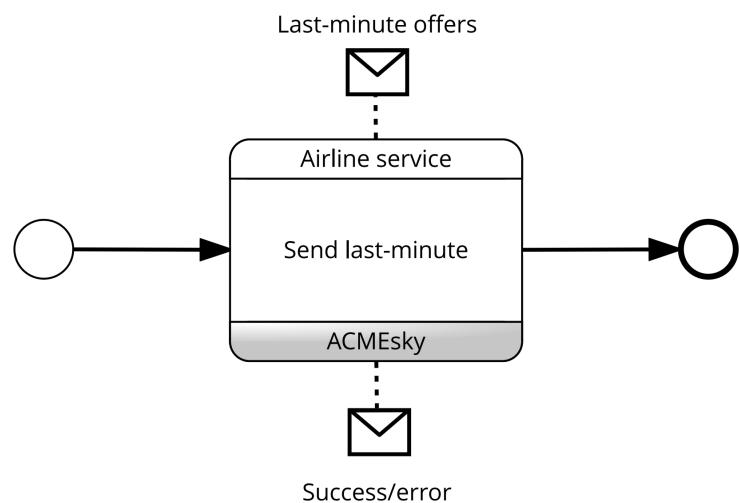
La coreografia descrive come l'utente e *ACMEsky* interagiscono nello scenario dell'invio degli interessi. L'utente manda una User Interest, mentre *ACMEsky* risponde con un messaggio di corretto inserimento o di errore.

Ricerca dei voli di interesse



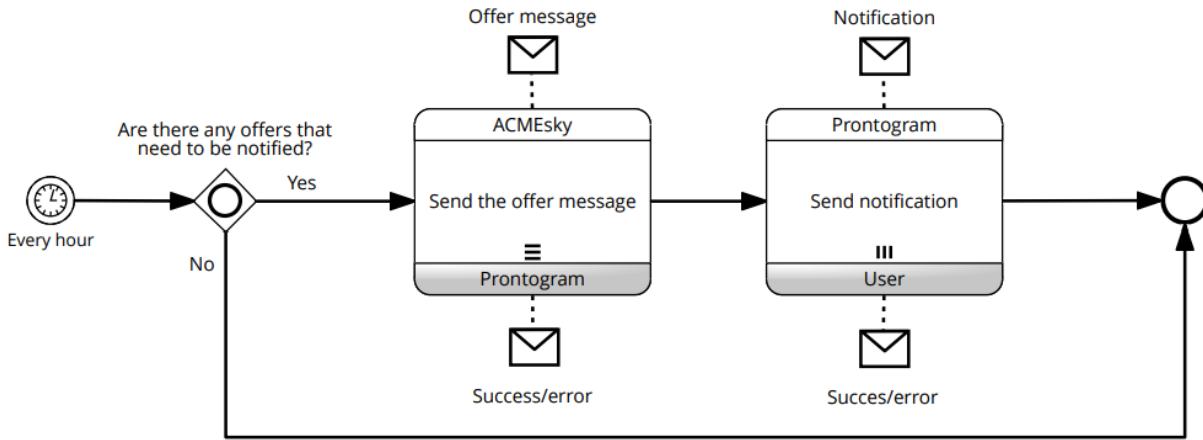
La coreografia descrive l'interazione tra *ACMEsky* e *Airline Service* al fine di cercare i voli che hanno una corrispondenza con quelli richiesti dagli utenti. Ogni ora e per ciascuna *Airline Service* registrata al servizio, *ACMEsky* recupera i voli di interesse degli utenti e effettua una richiesta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

Ricezione dei voli last-minute



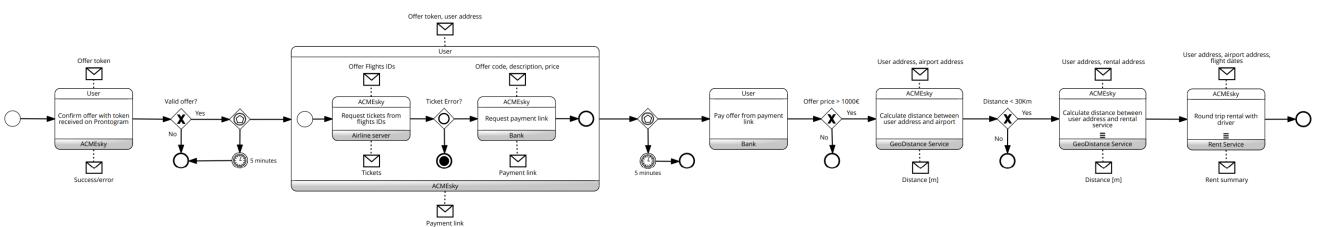
La coreografia descrive come *ACMEsky* e *Airline Service* interagiscono nello scenario della ricezione e salvataggio dei voli last-minute. *Airline Service* invia i voli last-minute ad *ACMEsky* che risponde con un messaggio di corretto inserimento o di errore.

Generazione dell'offerta



La coreografia mostra come *ACMEsky*, *Prontogram* e l'utente si relazionano al fine di notificare l'utente della corretta generazione dell'offerta. Ogni ora, se ci sono delle nuove offerte per l'utente, *ACMEsky* manda un messaggio a *Prontogram* che a sua volta risponde con un messaggio di corretto inserimento o di errore. Successivamente *Prontogram* manda una notifica all'utente.

Conferma e acquisto dell'offerta

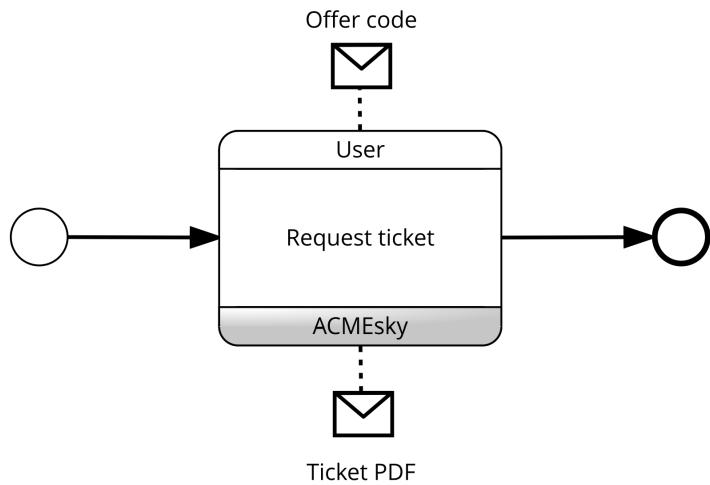


La coreografia descrive come l'utente, *ACMEsky*, *Bank Service*, *Airline Service*, *GeoDistance Service* e *Rent Service* interagiscono nel contesto della conferma e acquisto dell'offerta di volo da parte dell'utente. Quest'ultimo, conferma la volontà di voler acquistare l'offerta proposta inserendo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

ACMEsky verifica che l'offerta sia ancora valida, se non lo è il flusso termina. Altrimenti, il processo continua fino alla sotto-coreografia, 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 gli restituisce i biglietti. Se l'offerta è ancora disponibile per l'acquisto *ACMEsky* procede con il recupero del link del pagamento mandando un messaggio 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 fornитogli 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'aeroporto di partenza, inviando i rispettivi indirizzi al servizio per il calcolo delle distanze che restituirà la distanza tra i due punti. Se questa è superiore ai 30Km, ACMEsky si servirà nuovamente di *GeoDistance Service* per trovare la compagnia di noleggio più vicina. Infine, viene prenotato il trasporto per l'andata e per il ritorno dall'aeroporto attraverso la *Rent Service* individuata.

Recupero del biglietto



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

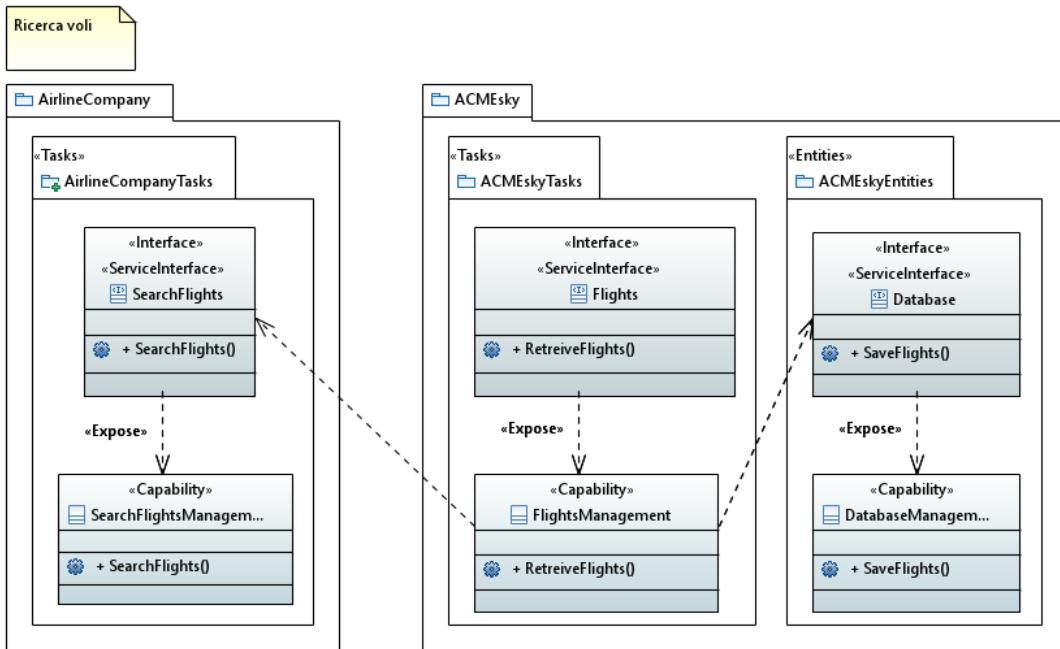
Diagrammi UML

In questa sezione 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 (Service Oriented Architecture), 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:

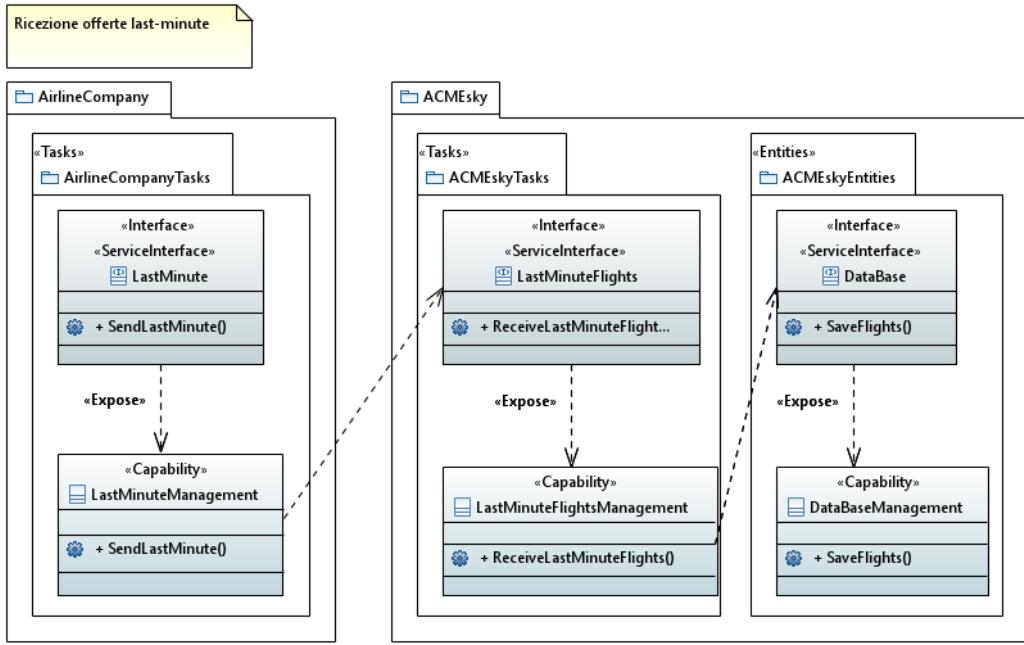
- **Task**: espone le **capability** facenti parte dei processi interni all'organizzazione;
- **Entity**: fa riferimento ad una singola attività, possibilmente automatizzata;
- **Utility**: sono simili ai task, però non appartengono al dominio del problema.

Ricerca voli



Nel diagramma riportato qui sopra vengono descritte le **capability** inerenti alla ricerca voli. In particolare, per il ruolo di ACMEsky sono definite 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.

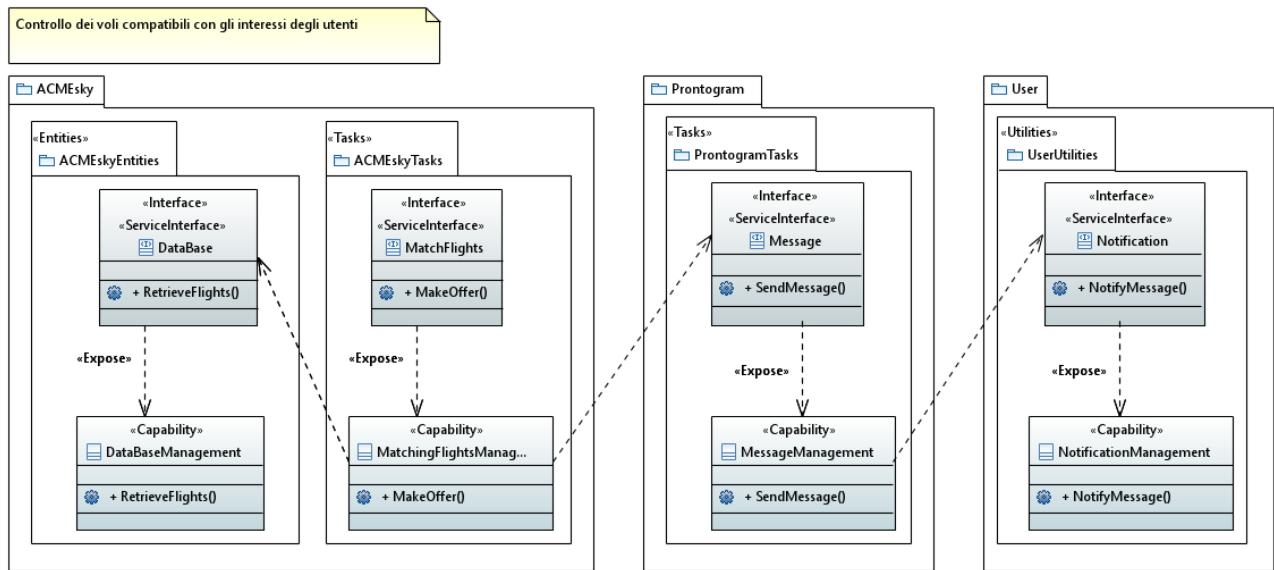
Ricezione offerte last-minute



Nel diagramma riportato qui sopra vengono descritte le capability inerenti alla ricezione delle offerte last-minute. In particolare per il ruolo di ACMEsky sono definite 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

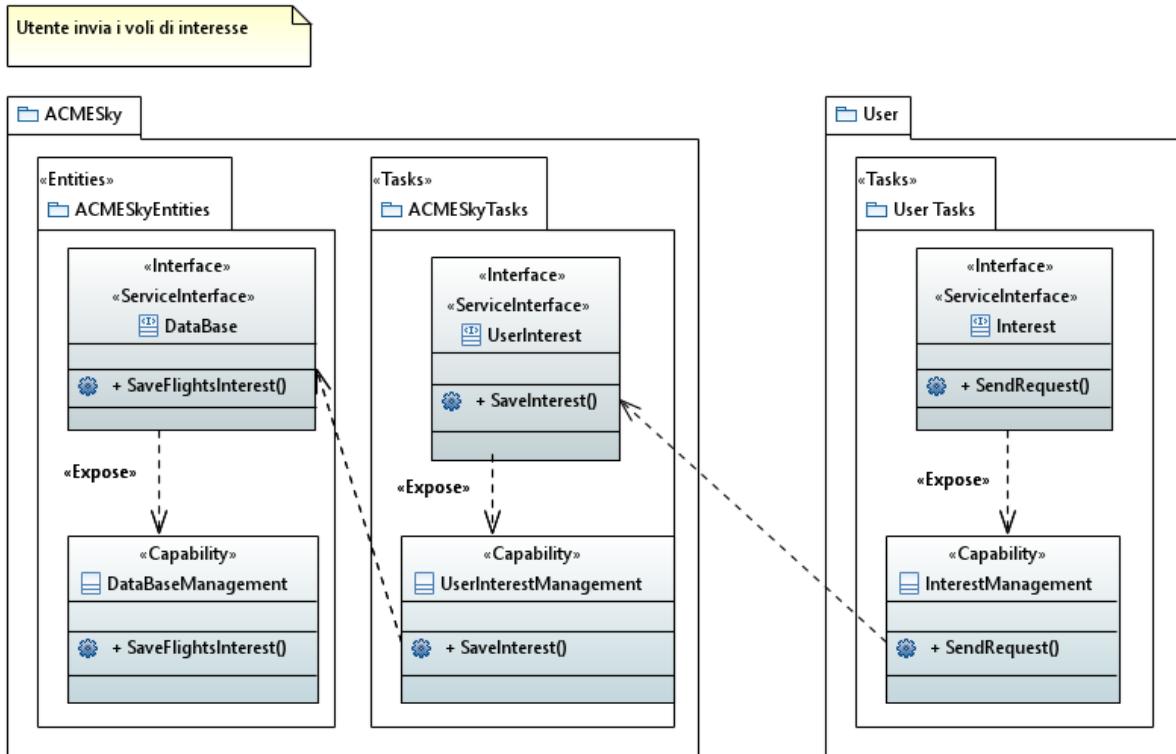


Nel diagramma riportato qui sopra vengono descritte le capability inerenti alla ricezione delle offerte last-minute. In particolare per il ruolo di ACMEsky sono definite le seguenti **capability**:

MatchingFlightsManagement e DataBaseManagement; Le quali vengono esposte da due interfacce **HourlyMatchFlights** e **DataBase**. La capability **MatchingFlightsManagement**, dopo avere reperito i voli

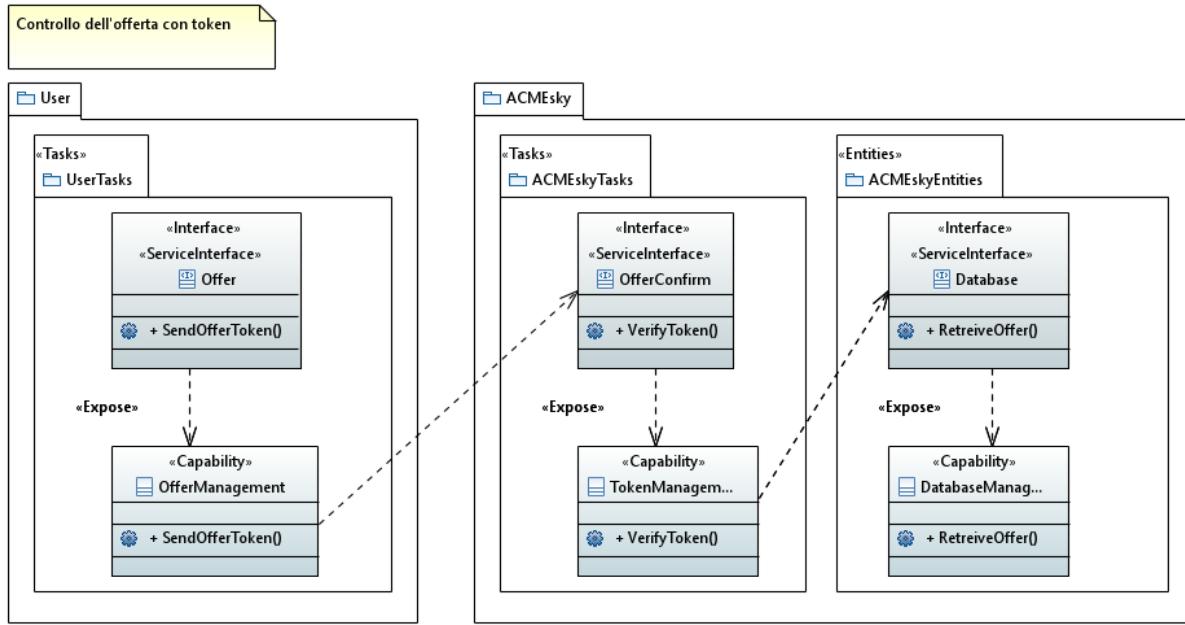
dalla base dati attraverso **DatabaseManagement**, ha lo scopo di trovare l'offerta compatibile con quella di interesse per l'utente. Inoltre la capability **MatchingFlightsManagement** dipende dalla capability di invio messaggi di Prontogram, **MessageManagement** esposta dalla relativa interfaccia: **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



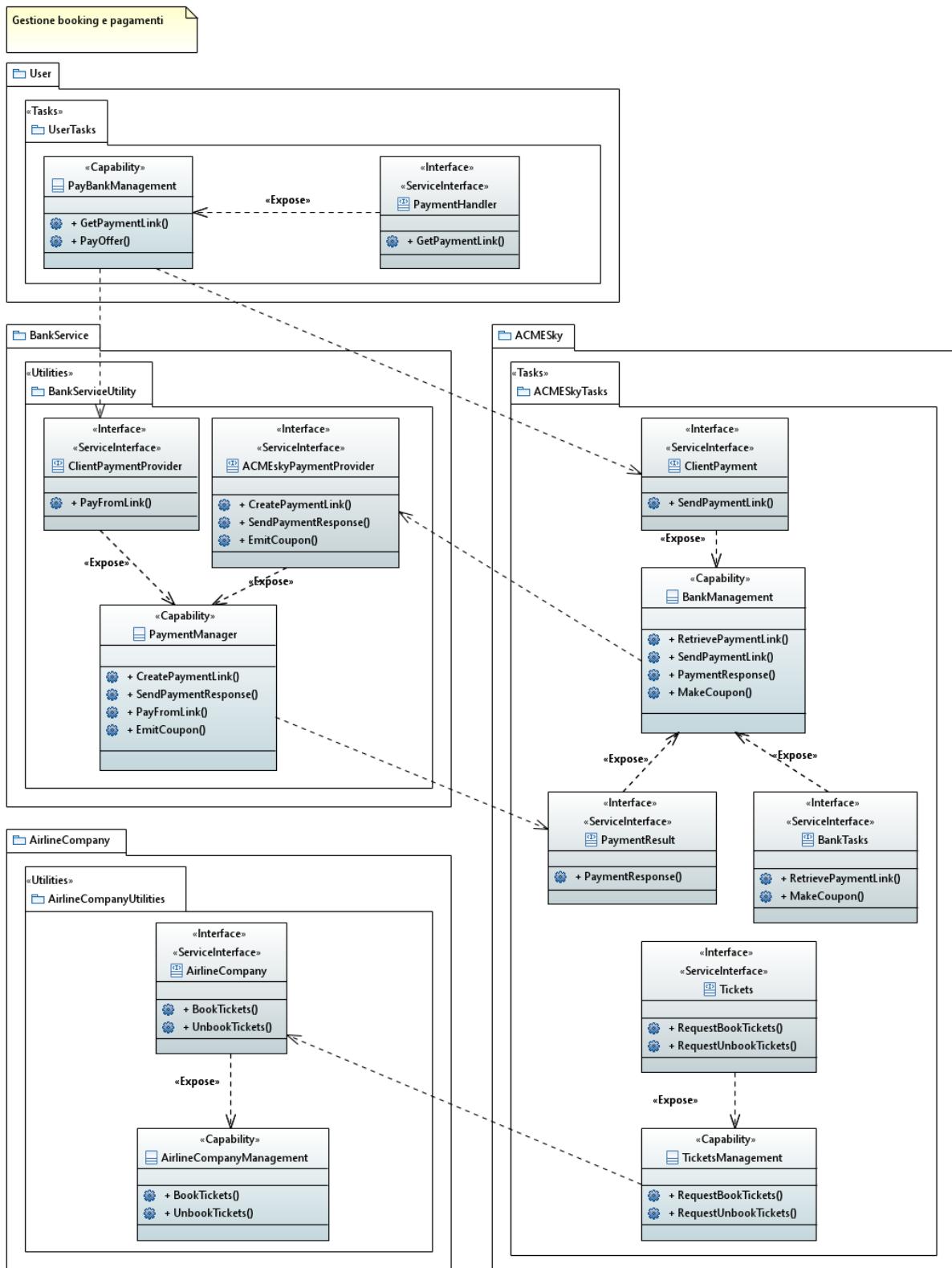
Nel diagramma riportato qui sopra vengono descritte le capability inerenti all'invio da parte di un utente della richiesta di un volo. In particolare per il ruolo di ACMESky sono definite le seguenti **capability**: **UserInterestManagement** e **DatabaseManagement**; Le quali vengono esposte da due 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 **UserRequestManagement**, 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 capability inerenti al controllo dell'offerta di interesse attraverso l'utilizzo del token inviato dall'utente. In particolare per il ruolo di ACMEsky sono definite 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.

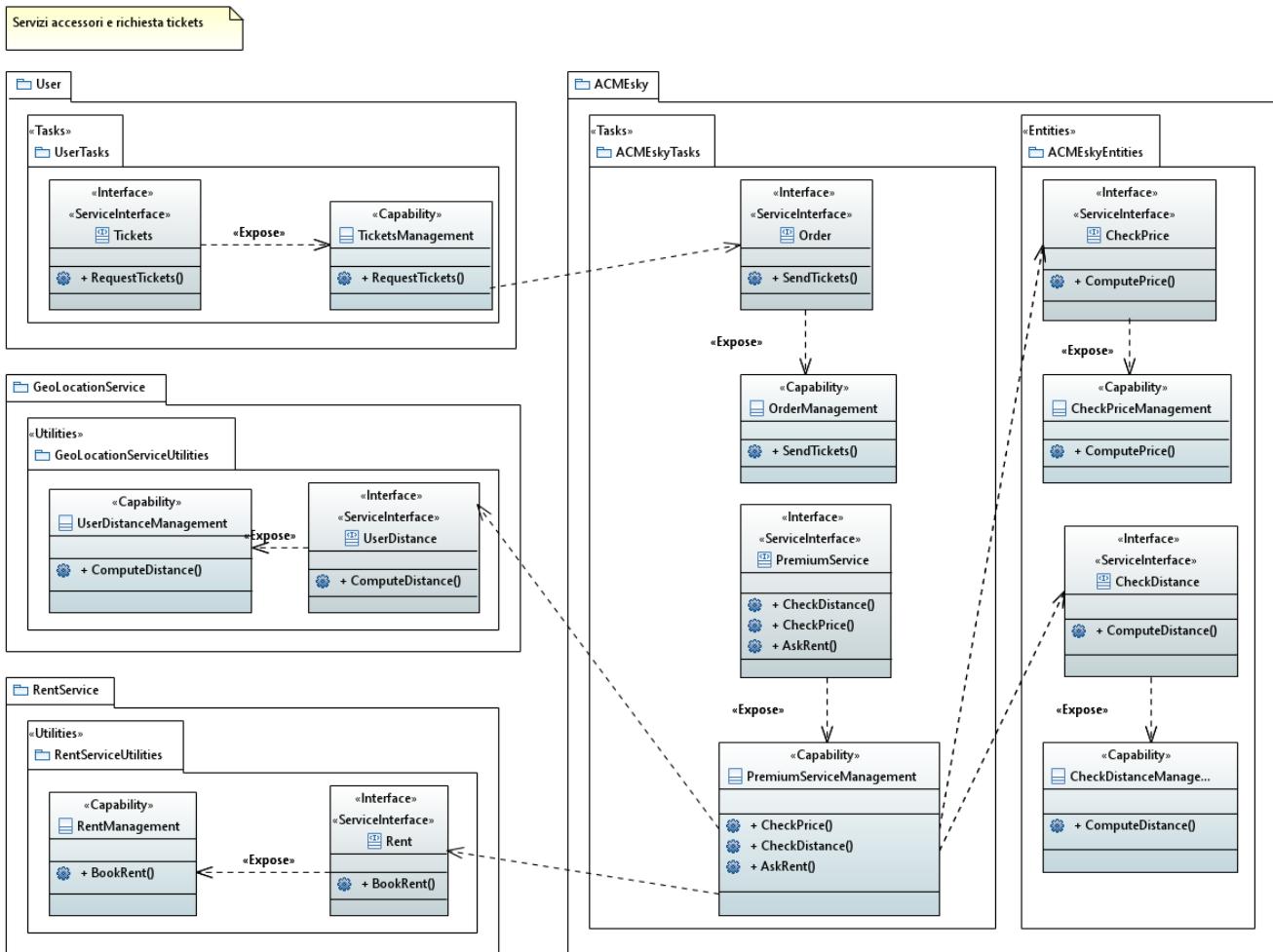
Gestione dei pagamenti



Nel diagramma riportato qui sopra vengono descritte le capability inerenti alla gestione dei pagamenti effettuati dall'utente indirizzati ai vari servizi. In particolare per il ruolo di ACMESky sono definite le seguenti **capability: BankManagement e TicketsManagement**; Le quali vengono esposte da un'unica interfaccia **Bank** e **Tickets**. 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. La capability **TicketsManagement** si occupa di prenotare il volo o di eliminare la prenotazione in caso di errori.

Servizi accessori e ricezione tickets

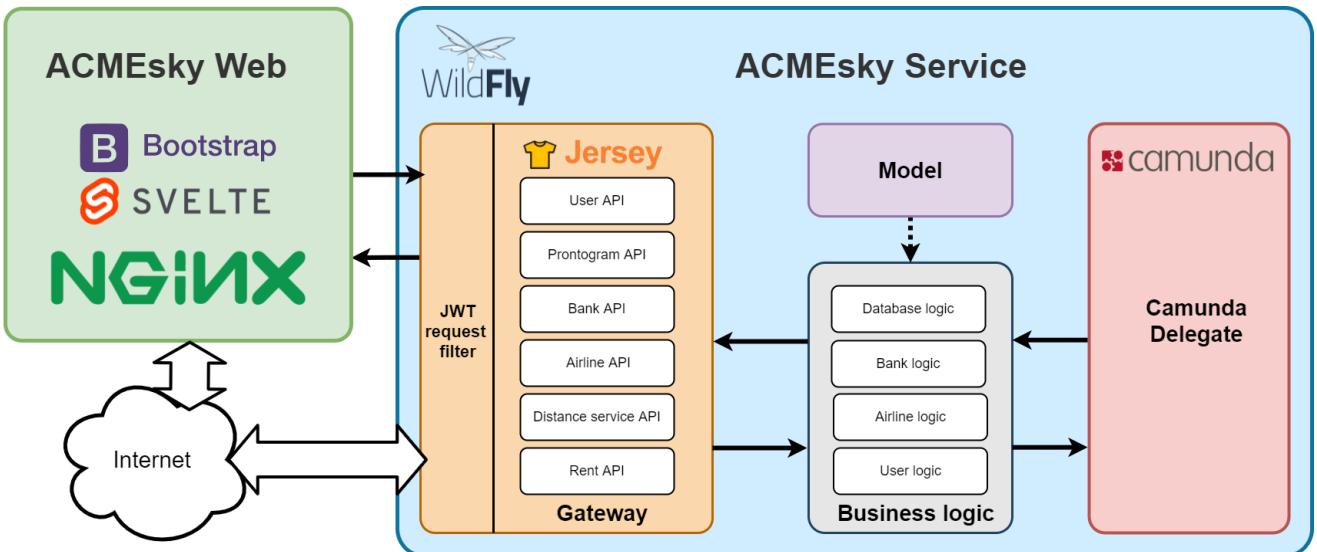


Nel diagramma riportato qui sopra vengono descritte le capability inerenti ai servizi accessori e ricezione dei tickets di interesse. In particolare per il ruolo di ACMEsky sono definite le seguenti **capability**:

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, ossia controlla il prezzo dell'offerta, la distanza dall'aeroporto e l'eventuale prenotazione di un servizio di trasporto. Infine la capability **OrderManagement** ha il compito di inviare i tickets richiesti dall'utente.

Struttura del sistema

In questa sezione si presenta la struttura interna di ACMEsky.

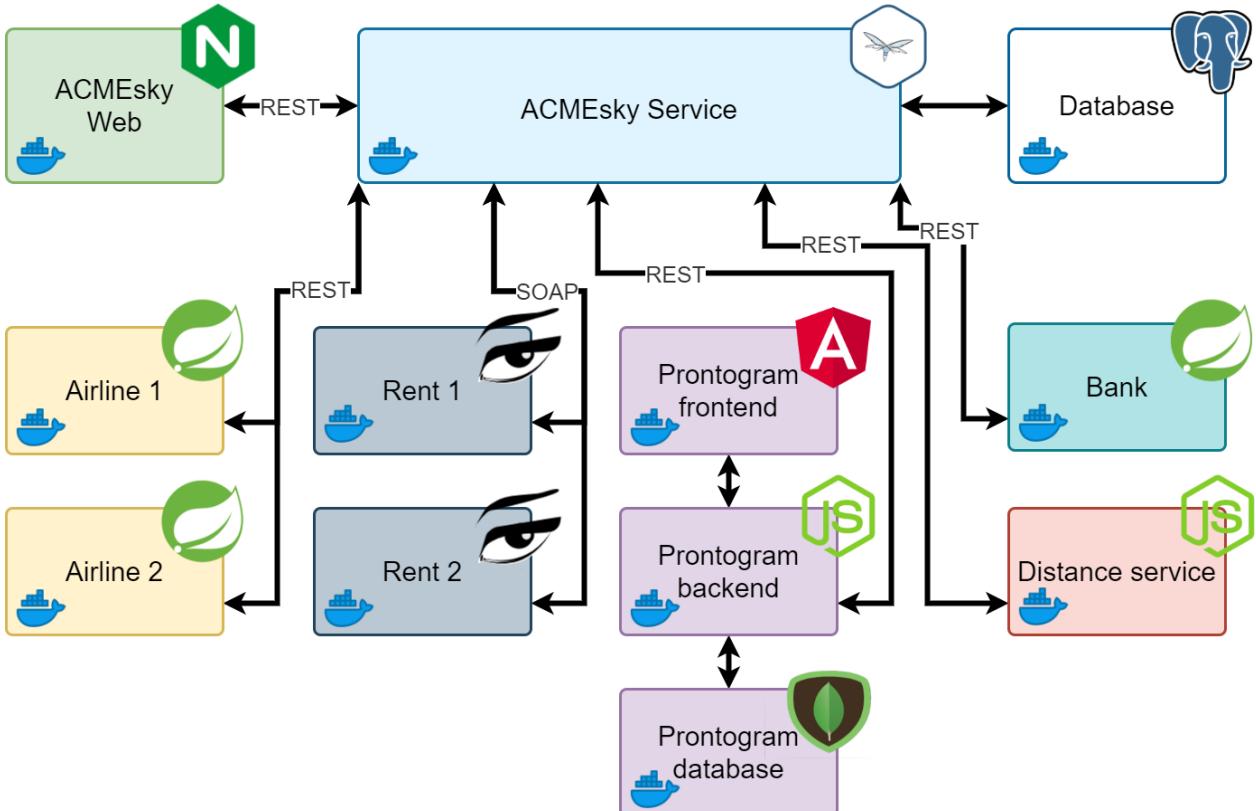


ACMEsky è stato sviluppato utilizzando *Java Enterprise Edition*. Per il deployment si è scelto *WildFly*, un application server open source multipiattaforma. Come BPMS si è optato per *Camunda* che fornisce un deployment già configurato per *WildFly*. I dati vengono gestiti da un database esterno al sistema.

ACMEsky, internamente, è stato suddiviso in più moduli che rispecchiano la struttura del codice, ognuno dei quali svolge un ruolo specifico nel sistema. Essi sono:

- **Camunda Delegate:** Sono le classi Java che implementano le funzionalità dei task nei processi BPMN.
- **Model:** Contiene le classi che rappresentano i dati utilizzati da ACMEsky. Attraverso l'utilizzo del framework JPA (Java Persistence API) le classi del *Model* vengono mappate nelle tabelle del database in maniera automatica.
- **Business logic:** In essa sono contenute tutte le classi Java che, utilizzando il *Model*, si occupano di gestire la logica business dei servizi di ACMEsky. Inoltre, si interfaccia con il DBMS per manipolare e archiviare i dati che vengono elaborati.
- **Gateway:** Raggruppa tutte le classi che implementano le API RESTful e SOAP di ACMEsky. La Business logic comunica con il *Gateway* per inviare e ricevere i messaggi dall'esterno. Inoltre, attraverso il filtro *JWT*, il *Gateway* autentica l'entità che effettua la richiesta e successivamente la autorizza a compiere operazioni su specifiche risorse.
- **ACMEsky Web:** È l'applicazione web di ACMEsky, essa permette all'utente di eseguire le richieste alle API di ACMEsky Service in maniera intuitiva. ACMEsky Web non è strettamente necessario per effettuare le richieste ai servizi esposti da ACMEsky, in quanto quest'ultimo potrebbe essere interrogato attraverso un semplice client REST.

Diagramma dei servizi



Per simulare un ambiente reale, ogni servizio, o parte de esso, è stato "incapsulato" in un container docker. Il diagramma mostra i container che vengono creati utilizzando il file `docker-compose.yml` che si trova nella root della repository del progetto. Di seguito vengono spiegati brevemente i servizi e la loro cunicazione con ACMEsky.

- **ACMEsky**
 - **ACMEsky Service:** Il server WildFly, che comprende il deployment di ACMEsky e Camunda BPMS, è stato inserito in un container. Esso si interfaccia con un altro container che contiene il database.
 - **ACMEsky Web:** L'applicazione web gira su un server NGNIX e si interfaccia ad ACMEsky attraverso le sue API RESTful.
- **Airline Service:** E' il servizio che simula una compagnia aerea, realizzato utilizzando il framework Spring. I servizi di Airline Service sono due: uno che offre voli nazionali e uno che offre voli internazionali, modificando il file `docker-compose.yml` è possibile aggiungerne di nuovi o rimuovere quelli già presenti. Essi si interfacciano ad ACMEsky utilizzando le API RESTful.
- **Bank Service:** E' il servizio con cui ACMEsky si interfaccia per la gestione simulata dei pagamenti, realizzato utilizzando il framework Spring. Viene ospitato in un container e comunica con ACMEsky attraverso le API RESTful.
- **GeographicalDistanceService:** E' il servizio di geolocalizzazione che calcola la distanza tra due indirizzi o coordinate geografiche. E' scritto in NodeJS utilizzando la libreria ExpressJS. Viene ospitato in un

container e comunica con ACMEsky attraverso API RESTful.

- **Prontogram:** Applicazione web che permette all'utente di ricevere le notifiche delle offerte dei voli generate da ACMEsky. Il front-end è stato realizzato utilizzando Angular, mentre il back-end è stato sviluppato utilizzando NodeJS. Comunica con ACMEsky attraverso API RESTful.
- **Rental Service:** E' il servizio di noleggio con autista. E' stato realizzato in Jolie. Comunica con ACMEsky attraverso SOAP. Nel deployment sono presenti due copie del servizio ciascuna distribuita all'interno di un container.

Porte assegnate ai servizi nel file `docker-compose.yml`

Service	Port
ACMEsky Service	8080
ACMEsky Web	80
Postgres	5050
BankService	8070
AirlineNationalService	8060
AirlineService	8061
ProntogramService	8050
ProntogramFrontEnd	8051
GeographicalService	8040
RentService1	8030
RentService2	8032

Istruzioni per l'esecuzione dello stack di container

Prerequisiti

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

Esecuzione

Dalla root della repository eseguire

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

I singoli servizi non devono essere compilati poiché questa operazione viene già fatta a tempo di build all'interno dei container.

Descrizione dei servizi

Di seguito vengono spiegati nel dettaglio i servizi.

ACMEsky Service

ACMEsky Service è il modulo principale di ACMEsky, si relaziona con i vari servizi (AirlineServices, RentServices, Prontogram, BankService, ecc...) al fine di consentire agli utenti di richiedere ed acquistare le offerte dei voli che desiderano.

ACMEsky Service comunica con il database (PostgreSQL) per gestire le entità che possono essere utenti o altri servizi, in modo da poterli autenticare e autorizzare quando inviano richieste HTTP. Inoltre, gestisce i voli di interesse degli utenti (A/R), i voli delle compagnie aeree (last-minute e non) e le offerte generate dai voli acquisiti dalle compagnie aeree.

Esso interagisce con ACMEsky Web per rapportarsi con gli utenti, raccoglie i voli di interesse degli utenti. Al momento dell'inserimento del token per acquistare l'offerta verifica se è ancora valida e procede con l'acquisto e il pagamento dei biglietti aerei. Infine, 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 che coincidono con gli interessi degli utenti. Gli interessi degli utenti sono composti da voli di andata e ritorno, mentre le offerte generate sono quelle che ACMEsky crea con i voli reali che riceve dalle compagnie aeree. Inoltre, il servizio riceve voli last-minute inviati dalle Airline Services con una certa frequenza. Infine, ACMEsky recupera i biglietti dei voli che gli utenti vogliono acquistare tramite una richiesta alla compagnia aerea, nel caso di errori o problemi di pagamento cancella la prenotazione informando l'AirlineService corrispondente.

Per quanto riguarda il pagamento, il servizio interroga Bank Service (provider di pagamenti), per richiedere il link di pagamento da inviare all'utente, il quale poi interagirà con la banca per effettuare il pagamento. Infine, Bank Service informerà ACMEsky dell'avvenuto pagamento.

Il servizio di ACMEsky interagisce con quello del calcolo delle distanze geografiche "GeographicalDistance Service" e con quelli di noleggio "Rental Service" per applicare eventuali servizi aggiuntivi all'offerta acquistata dell'utente. Effettua le richieste a GeographicalDistance Service per calcolare la distanza utente - aeroporto e per trovare la compagnia di noleggio più vicina. Infine, prenota il trasferimento A/R da Rental Service e aggiunge i dettagli sulla ricevuta di viaggio.

Tecnologie utilizzate e scelte progettuali

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 l'application server Wildfly che offre supporto completo a Java EE in tutti gli ambienti applicativi. E' stato utilizzato Camunda come BPMN per supportare i processi, il quale offre un deployment per Wildfly. Il servizio mette a disposizione 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. Il deployment di ACMEsky è basato sull'immagine Docker `camunda-bpmn-platform:wildfly` a cui viene aggiunto il file .war compilato dai sorgenti di ACMEsky.

Il progetto è composto dai seguenti moduli:

camunda

Questa parte del progetto si compone di tre sottodirectory, ovvero flights_manager, offers_manager e user_manager: questo perchè si è voluto riprendere la struttura del diagramma BPMN, che divide i vari flussi di esecuzione dei processi in queste tre lane che si differenziano per la loro funzione e per gli attori con cui interagiscono. Ciascuna classe implementa un task di un processo o sottoprocesso presente nel flusso di esecuzione, implementando l'interfaccia JavaDelegate e definendo il metodo execute.

- **flights_manager** si suddivide nelle seguenti directory: *last_minute* che ospita al proprio interno un file che consente di salvare le offerte che le compagnie aeree inviano ad ACMEsky, *remove_expired_flights* la quale include i file per rimuovere i voli scaduti e *search_flights* che consente di recuperare la lista degli AirlineService, cercare i voli di interesse e salvarli in db.
- **offers_manager** comprende i Delegate per rimuovere le offerte di volo e relativi voli scaduti, scegliere tra i voli disponibili quelli che corrispondono agli interessi degli utenti, preparare l'offerta e inviarla all'utente.
- **user_manager** si suddivide in: *book_payment*, che gestisce la prenotazione e l'acquisto dei biglietti aerei, *confirm_offer*, il quale si occupa della conferma dell'offerta espressa dall'utente inserendo il token e controllando che l'offerta sia ancora valida e *premium_service*, controlla l'applicazione di eventuali servizi aggiuntivi all'offerta. Infine *save_interest* si occupa del salvataggio dell'offerta di interesse degli utenti. Gli altri file sono legati ai task finali relativi al 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.

Le classi presenti nella directory utils definiscono: le variabili dei processi, gli eventi di inizio e gli eventi di errore.

gateway

Questa parte del progetto si compone di una directory per ciascun servizio, nelle quali si descrivono le interfacce esposte ad essi. Inoltre, ACMEsky implementa le api dei servizi esterni che verranno utilizzate dalla business logic.

business logic

Ospita i manager, che utilizzando i modelli, implementa la business logic di ACMEsky. In particolare, interroga il database e attraverso il gateway comunica con i servizi esterni. Ciascuna classe si occupa della

gestione dei servizi corrispondenti. E' composto da AirlineManager, BankManager, InterestManager e OfferManager.

model

Describe i dati coinvolti nel progetto: le entità (utente, AirlineService, BankService e RentService), i voli ricevuti dalle compagnie aeree e quelli di interesse, le offerte di interesse e quelle generate da ACMEsky, e gli aeroporti.

security

Si occupa dell'autenticazione e dell'autorizzazione delle entità che fanno richieste attraverso le api di ACMEsky. Attraverso la route `/auth` è possibile autenticarsi e richiedere il token JWT, con il quale dovranno essere effettuate le successive richieste. Attraverso il token delle richieste viene consentito ai servizi di accedere alle risorse autorizzate per il proprio ruolo.

utils

Contiene le classi che descrivono gli errori restituiti in caso di problemi nelle richieste, le variabili d'ambiente e quella di utilità per i pdf dei biglietti.

Formato ricevuta



API

Il file OpenAPI è disponibile al seguente [link](#)

Risorse di autenticazione

Risorsa	Descrizione
POST /auth	Questa risorsa consente di autenticarsi nel sistema. Si richiede un oggetto AuthRequestDTO come parametro, composto da un attributo username e un attributo password. ACMEsky restituirà un token valido.
PUT /auth/refresh	Questa risorsa consente alle entità di richiedere un nuovo token dato uno che sta per scadere.

Risorse per AirlineServices

Risorsa	Descrizione
POST <code>/airlines/last_minute</code>	Questa risorsa è riservata esclusivamente agli AirlineServices. La chiamata a questa risorsa richiede come parametro una lista di oggetti AirlineFlightOfferDTO che verrà salvata nel database.

Risorse per BankServices

Risorsa	Descrizione
GET <code>/bank/confirmPayment</code>	Questa risorsa è riservata esclusivamente ai servizi BankServices. La chiamata a questa risorsa richiede come parametro il codice dell'offerta di volo acquistata dall'utente.

Risorse per ACMEskyWeb

Risorsa	Descrizione
GET <code>/airports</code>	Questa risorsa è riservata esclusivamente all'utente, esso accede a questa risorsa passando la query di ricerca dell'aeroporto per recuperare la lista dei suggerimenti.
GET <code>/airports/{code}</code>	Questa risorsa è riservata esclusivamente all'utente, restituisce l'aeroporto associato al codice IATA fornito come parametro.
POST <code>/interests</code>	Questa risorsa è riservata esclusivamente all'utente, e permette di inserire gli interessi specificati nell'oggetto UserInterestDTO.
GET <code>/interests</code>	Questa risorsa è riservata esclusivamente all'utente. Consente di recuperare la lista di tutte le offerte di interesse dell'utente che si è autenticato sulla piattaforma ACMEskyWeb.
GET <code>/interests/{id}</code>	Questa risorsa è riservata esclusivamente all'utente. Consente di recuperare l'offerta di interesse corrispondente all'identificativo passato come parametro del path della richiesta.
DELETE <code>/interests/{id}</code>	Questa risorsa è riservata esclusivamente all'utente. Permette di cancellare l'offerta di interesse con lo stesso identificativo di quello passato come parametro del path della richiesta.
PUT <code>/offers/confirm</code>	Questa risorsa è riservata esclusivamente all'utente. 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	Questa risorsa è riservata esclusivamente all'utente. Consente di

/offers/paymentLink	recuperare il link di pagamento passando il DTO dell'indirizzo dell'utente come parametro.
PUT /offers/reset	Questa risorsa è riservata esclusivamente all'utente. Consente di fare il reset del processo di conferma e acquisto dell'offerta passando come parametro un oggetto UserInterestDTO dell'utente.
GET /offers/	Questa risorsa è riservata esclusivamente all'utente. Restituisce 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 all'utente. Recupera l'offerta generata con il token corrispondente a quello specificato.
GET /offers/{token}/ticket	Questa risorsa è riservata esclusivamente all'utente. Restituisce il biglietto dell'offerta con il token specificato dal parametro.
GET /users/me	Questa risorsa è riservata esclusivamente all'utente. Recupera le informazioni dell'utente che si è autenticato.
POST /users/	Tutti i nuovi servizi e utenti possono effettuare chiamate a questa route per registrarsi su ACMEsky. Il servizio chiamante può effettuare chiamate a questa route passando come argomento un oggetto di tipo UserSignUpUpdDTO, contenente email, password, name, surname e prontogramUsername, per registrare un utente.
DELETE /users/me	Questa risorsa è riservata esclusivamente all'utente. Permette di cancellare l'utente dal db di ACMEsky.

comando per fare la build

```
mvn package
```

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

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

ACMEsky Database

ACMEsky si interfaccia ad un database PostgreSQL. Il file docker-compose, attraverso il file init.sql, contiene tutto il necessario per eseguire un'istanza del database con le tabelle e i record necessari ad ACMEsky.

Database schema

airports	
PK	<i>id</i>
	<i>code</i>
	<i>name</i>
	<i>city_name</i>
	<i>country_code</i>
	<i>timezone</i>
	<i>latitude</i>
	<i>longitude</i>

La tabella ***airports*** contiene i record che rappresentano gli aeroporti nazionali e internazionali codificati secondo il codice IATA. Essa contiene il campo ***id*** (chiave primaria), ***code*** (codice dell'aeroporto in codifica IATA), ***name*** (nome dell'aeroporto), ***city_name*** (nome della città), ***country_code*** (codice del paese), ***timezone*** (fuso orario), ***latitude*** e ***longitude*** (la posizione dell'aeroporto).

domain_entities	
PK	<i>id</i>
	<i>username</i>
	<i>password</i>
	<i>salt</i>
	<i>role</i>

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** (dato random addizionato all'input della funzione one-way), e **role** (il ruolo dell'entità nella SOA).

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** (chiave primaria), **user_id** (chiave esterna della tabella **users**), **departure_airport_id** e **arrival_airport_id** (indicano l'identificatore dell'aeroporto di partenza e arrivo), **departure_date_time** (data di partenza) e **used** (indica se ACMEsky ha già proposto all'utente il volo in un'offerta).

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 i voli di interesse degli utenti A/R. Comprende i campi: ***id*** (chiave primaria), ***user_id*** (chiave esterna dell'utente nella tabella ***users***), ***outbound_flight_interest_id*** (chiave esterna del volo di interesse di andata in ***flights_interest***), ***flight_back_interest_id*** (chiave esterna del volo di interesse di ritorno in ***flights_interest***), ***price_limit*** (limite di prezzo che l'offerta non può superare), ***expire_date*** (data di scadenza entro cui l'offerta è prenotabile) e ***used*** (valore booleano che segnala se l'interesse è stata già gestita da ACMEsky).

	airlines
PK	<i>id</i>
FK	<i>entity_id</i>
	<i>ws_address</i>

La tabella ***airlines*** fa riferimento ai servizi delle compagnie aeree (AirlineService). Comprende i seguenti campi: ***id*** (chiave primaria), ***entity_id*** (chiave esterna dell'entità in ***domain_entities***) e ***ws_address*** (l'indirizzo del server del servizio a cui si possono fare richieste).

	flights
PK	<i>id</i>
FK	<i>departure_airport_id</i>
FK	<i>arrival_airport_id</i>
FK	<i>airline_id</i>
	<i>flight_code</i>
	<i>departure_date_time</i>
	<i>arrival_date_time</i>
	<i>price</i>
	<i>expire_date</i>
	<i>booked</i>

available

La tabella ***flights*** descrive i voli che vengono recuperati interrogando la compagnia aerea sulla base dei voli di interesse degli utenti. Sono presenti i seguenti campi: ***id*** (chiave primaria), ***departure_airport_id*** (chiave esterna dell'aeroporto di partenza nella tabella ***airports***), ***arrival_airport_id*** (chiave esterna dell'aeroporto di arrivo nella tabella ***airports***), ***airline_id*** (chiave esterna della compagnia aerea nella tabella ***airlines***), ***flight_code*** (codice con il quale il volo viene registrato dalla compagnia), ***departure_date_time*** (data e ora di partenza del volo), ***arrival_date_time*** (data e ora di arrivo), ***price*** (prezzo del volo), ***expire_date*** (data di scadenza del volo, ossia quando non è più prenotabile), ***booked*** (indica se il volo è stato prenotato) e ***available*** (stabilisce se il volo è già stato inserito in un offerta, quindi non più inutilizzabile).

generated_offers	
PK	<i>id</i>
FK	<i>user_id</i>
FK	<i>outbound_flight_id</i>
FK	<i>flight_back_id</i>
	<i>expire_date</i>
	<i>total_price</i>
	<i>booked</i>
	<i>token</i>

La tabella ***generated_offers*** rappresenta le offerte di volo generabili da ACMEsky sulla base degli interessi degli utenti. E' composta dai seguenti campi: ***id*** (chiave primaria), ***user_id*** (chiave esterna dell'utente nella tabella ***users***), ***outbound_flight_id*** (chiave esterna del volo di andata nella tabella ***flights***), ***flight_back_id*** (chiave esterna del volo di ritorno nella tabella ***flights***), ***total_price*** (prezzo dell'offerta A/R), ***expire_date*** (data di scadenza dell'offerta, ossia quando non è più prenotabile), ***booked*** (indica se l'offerta è stata prenotata o meno), ***token*** (codice dell'offerta).

banks	
PK	<i>id</i>
FK	<i>entity_id</i>
	<i>ws_address</i>

La tabella ***banks*** fa riferimento ai servizi bancari. E' composta dai seguenti campi: ***id*** (chiave primaria), ***entity_id*** (chiave esterna dell'entità della banca nella tabella ***domain_entities***) e ***ws_address*** (indirizzo del server del servizio).

rent_services	
PK	<i>id</i>
FK	<i>entity_id</i>
	<i>address</i>
	<i>ws_address</i>

La tabella ***rent_services*** fa riferimento ai servizi di noleggio per accompagnare l'utente all'aeroporto. E' composta dai seguenti campi: *id* (chiave primaria), *entity_id* (chiave esterna riferita alla entità nella tabella ***domain_entities***) e *ws_address* (indirizzo del server del servizio).

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
- Bootstrap

Interfaccia grafica

Home

The screenshot shows the ACMESKY home page. At the top, there is a navigation bar with the text "ACMESKY" on the left and "SIGNIN" on the right. Below the navigation bar, the main content area has a title "BENVENUTO IN ACMESKY" and a sub-instruction "Se non l'hai ancora fatto, registrati!". Below this, a numbered list describes a process:

1. INSERISCI LE TUE TRATTE DI INTERESSE
2. RICEVERAI LA TUA OFFERTA RISERVATA SU PRONTOGRAM™
3. ACCEDI E ACQUISTA L'OFFERTA
4. GODITI IL TUO VIAGGIO :)

Each step is preceded by a downward-pointing arrow.

Aggiunta interesse

AGGIUNGI VOLI DI INTERESSE

ANDATA E RITORNO

Partenza

Arrivo

Data di partenza

Data di ritorno

Prezzo limite €

SALVA

Conferma e acquisto offerta

ACQUISTA UN'OFFERTA

Offerta

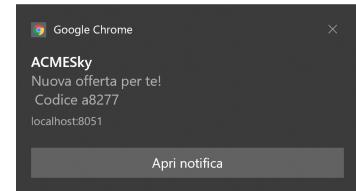
Informazioni

Pagamento

Riepilogo

Codice offerta

AVANTI



Riepilogo

ACQUISTA UN'OFFERTA

Offerta

Informazioni

Pagamento

Riepilogo

IL TUO VIAGGIO È PRONTO!

CODICE OFFERTA	2d675
AREOPORTO DI PARTENZA	Bologna (BLQ)
AREOPORTO DI ARRIVO	Amsterdam (AMS)
ANDATA E RITORNO	Si
DATA E ORA ANDATA	28/10/2021 20:00:00

Esecuzione

Installare 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
```

Airline service

Airline Service è il servizio che simula la compagnia aerea. Attraverso le sue API permette di cercare i voli nel database e di acquistarne i biglietti. Inoltre, ad intervalli regolari, genera i voli last-minute e li invia ad ACMEsky.

Per simulare un ambiente reale è stato creato un file `docker-compose.yml` che contiene due istanze di Airline, ovvero `national_airline` e `international_airline`. La prima offre voli da e verso aeroporti nazionali, mentre la seconda offre voli da e verso aeroporti internazionali.

Tecnologie utilizzate e scelte progettuali

Il servizio è stato realizzato con il framework Spring boot che fornisce un ambiente per sviluppare applicazioni web JAVA e in particolare servizi REST.

Al primo avvio, viene prelevata la lista dei voli dal file JSON inserito nella directory `fileSampleOffers` e viene caricata nel database. Il file JSON è suddiviso due array: "*OFFERS*", la quale include i biglietti per i voli subito disponibili e "*LAST-MINUTE*" che racchiude i biglietti per i voli last-minute. Inoltre, è possibile aggiungere nuovi biglietti o rimuovere quelli già presenti integrando il file con nuovi oggetti.

La generazione dei biglietti per i voli last-minute viene effettuata ogni 10 minuti scegliendo casualmente un volo tra quelli presenti nell'array. Le offerte di volo vengono convertite automaticamente in nuovi oggetti "Volo" prima di essere inviati ad ACMEsky, per racchiudere solo le informazioni utili.

Prima di inviare le offerte last-minute Airline si deve autenticare da ACMEsky, in caso di esito positivo, riceve un token JWT che dovrà inserire nella richiesta di invio delle offerte.

Per la generazione dei biglietti aerei da inviare all'utente si è scelto di usare la libreria di Thymeleaf, che permette di generare file pdf partendo da un template in formato HTML e CSS riempito con i dati a runtime.

Per la persistenza dei dati è stato scelto H2, un leggero DBMS scritto in Java con tecnologia in-memory.

Struttura del codice

Classi principali

- **FlightOfferService:** Contiene i metodi per la creazione, ed il salvataggio delle offerte di volo nel database e per l'eventuale invio, nel caso siano last-minute. Inoltre, sono presenti i metodi per cercare le offerte di volo e per cambiare lo stato di acquisto dei biglietti.
- **PdfService:** Si occupa di generare i file pdf che contengono le informazioni sui biglietti.

Il package **model** include le classi per la definizione dell'offerta di volo (**FlightOffer**) e la classe per la definizione di alcune utility per la generazione e gestione delle offerte (**FlightUtility**).

Il package **DTO** include le classi 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 (**AuthRequest**).

La directory **resources** contiene i file necessari per realizzare i biglietti aerei.

API:

Il file OpenAPI è disponibile al seguente [link](#)

Risorse

Risorsa	Descrizione
POST /getFlights	Ricerca dei voli disponibili nel database. Prende in input una lista di oggetti <i>UserRequest</i> .
GET /getTickets	Acquista e restituisce i biglietti identificati dal parametro <i>id</i>

Esecuzione

Build fat Jar:

```
mvn package
```

Come eseguire con Docker compose

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

Credenziali database

DB console service 1

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

DB console service 2

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

URI

Gli URI riferiti dei container definiti in `docker-compose.yml`:

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

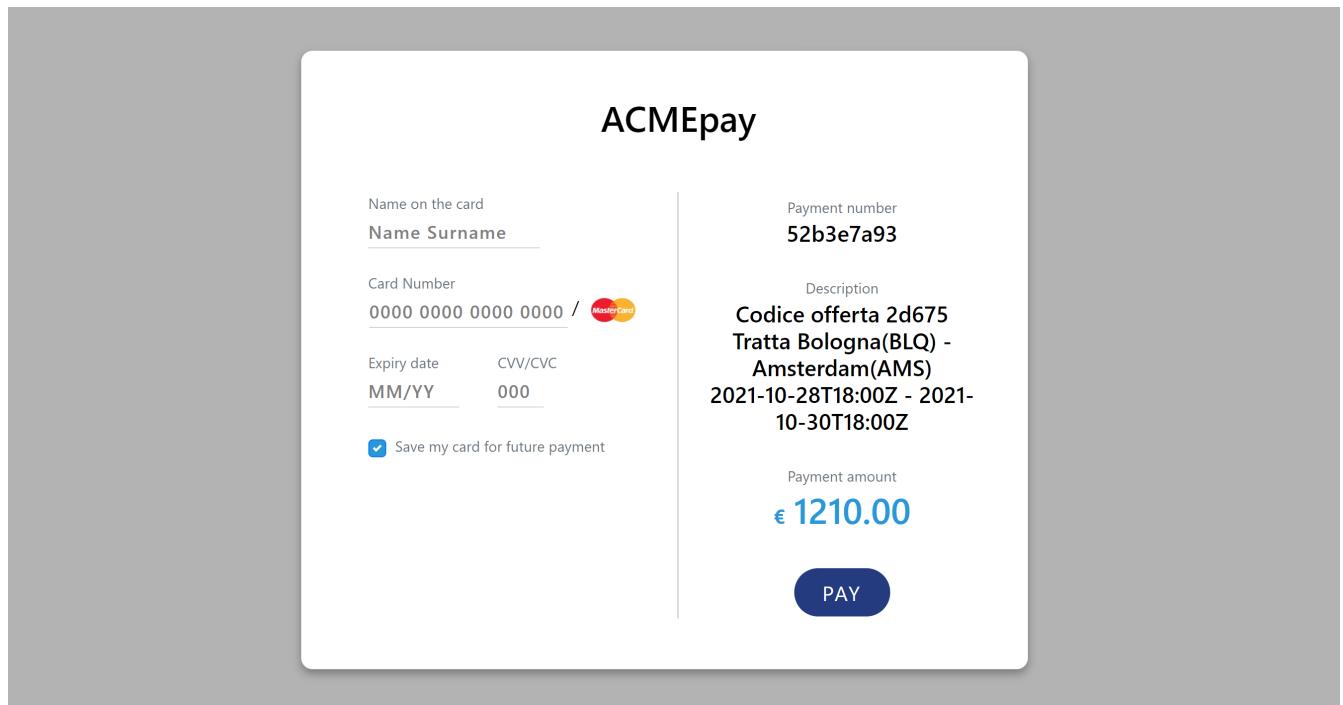
Bank service

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 essere autenticate utilizzando il token JWT che può essere richiesto utilizzando la route `/path`.

Tecnologie utilizzate e scelte progettuali

Il servizio è stato realizzato utilizzando il framework Spring boot che fornisce un ambiente per sviluppare applicazioni web JAVA. Per l'autenticazione degli utenti e l'autorizzazione delle richieste è stato utilizzata la libreria *Spring-Security* in accoppiata con *jsonwebtoken*. Per la gestione dei dati è stato utilizzato *H2*, un DBMS leggero che permette di salvare il database in un unico file.

Schermata di pagamento



API:

Il file OpenAPI è disponibile al seguente [link](#)

Risorse

Risorsa	Descrizione
POST /users	Risorsa che permette di creare un nuovo utente.
GET /users/me	Risorsa che restituisce le informazioni dell'utente loggato.
POST /payments	Risorsa che permette di generare un link di pagamento.
GET /payments	Risorsa che restituisce la lista dei link di pagamento associati all'utente.
POST /auth	Risorsa per autenticarsi nel sistema, restituisce un JWT che dovrà essere inserito in tutte le richieste.

Esecuzione

Build fat Jar:

```
mvn package
```

Come eseguire

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

Come eseguire con Docker Compose

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

Credenziali database

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

Prontogram

Home Gestione Utente Notifiche Logout

Notifiche

ACMEsky ha una nuova offerta per te!

Codice offerta 2d675
Areopporto di partenza (BLQ)
Areopporto di arrivo (AMS)
Data e ora andata 2021-10-28T18:00Z
Data e ora ritorno 2021-10-30T18:00Z
Prezzo totale 1210.00€

Elimina

ACMEsky ha una nuova offerta per te!

Codice offerta 31f36
Areopporto di partenza (BLQ)
Areopporto di arrivo (AMS)
Data e ora andata 2021-10-28T18:00Z

Prontogram è una applicazione web che permette di inviare notifiche agli utenti che vi sono iscritti. I messaggi inviati possono essere formattati utilizzando HTML, questo permette di rappresentare i dati in modo complesso (ad esempio in una tabella).

L'applicazione di Prontogram si divide in due parti: front-end e back-end. La parte back-end si occupa di gestire tutte le chiamate delle API da e verso il client web. Mentre la parte front-end si occupa di creare l'interfaccia grafica e gestire le interazioni da parte dell'utente.

Tecnologie utilizzate e scelte progettuali

La parte front-end e la parte back-end sono state sviluppate utilizzando tecnologie e pattern strutturali differenti.

Front-end

La parte front-end è stata realizzata utilizzando 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 della SPA (Single page application).

La parte front-end di Prontogram è formata da 3 micro-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 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 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 web. Il package *models* include la classe per la definizione della notifica (*notification*), la classe per la definizione dell'utente (*user*) e la classe per la definizione della sottoscrizione al server(*subscription*). Inoltre, è presente il package *routes*, il quale contiene tutti i 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

Il file OpenAPI è disponibile al seguente [link](#)

Risorse

Risorsa	Descrizione
POST <code>/notification/posts</code>	Esegue la creazione della notifica e il salvataggio della notifica nel database per poi inviarla a Prontogram attraverso il metodo <code>sendNotification()</code> . 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 <code>/notification/posts/{notificationId}</code>	Esegue la cancellazione di una notifica di un'offerta attraverso il passaggio come parametro dell'Id della notifica.
GET <code>/notification/gets/all/{username}</code>	Esegue il caricamento di tutte le notifiche delle offerte, attraverso il passaggio come parametro dell'username dell'utente corrispondente.
POST <code>/subscription/post/new</code>	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 <code>/subscription/posts/{subendpoint}</code>	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 <code>/subscription/gets/all</code>	La chiamata a questa route permette di reperire tutte le sottoscrizioni effettuate dagli utente al server.
POST <code>/user/posts/</code>	Esegue la creazione di un nuovo utente e il suo salvataggio nel database.
DELETE <code>/user/posts/{userId}</code>	Esegue l'eliminazione di un utente dal database, attraverso il passaggio come parametro dell'Id dell'utente corrispondente.
PUT <code>/user/posts/{userId}</code>	Esegue l'update delle informazioni relative ad un'utente, attraverso il passaggio come parametro dell'Id dell'utente corrispondente.
GET <code>/user/gets/{id}</code>	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 <code>/user/gets/all</code>	La chiamata a questa route permette di reperire tutte le informazioni relative a tutti gli utenti presenti in database.
POST <code>/auth/register</code>	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 <code>/auth/login</code>	Esegue il login da parte di un utente a Prontogram.

URI

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

## Esecuzione

---

### Come fare la build

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

### Come eseguire

```
cd static
http-server
```

### Eseguire con Docker Compose

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

# Rental service

Servizio che simula una compagnia di noleggio.

Implementato in Jolie, utilizza SOAP per esporre i servizi.

## Service ports

| Name            | Endpoint (Location)                                       |
|-----------------|-----------------------------------------------------------|
| RentServicePort | <a href="http://localhost:8080">http://localhost:8080</a> |

## PortType

| Operation | Input       | Output       |
|-----------|-------------|--------------|
| bookRent  | RentRequest | RentResponse |

## RentRequest

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:unib="www.unibo.it.xsd">
 <soapenv:Header/>
 <soapenv:Body>
 <unib:bookRent>
 <clientName>?</clientName>
 <clientSurname>?</clientSurname>
 <fromAddress>?</fromAddress>
 <arrivalDateTime>?</arrivalDateTime>
 <toAddress>?</toAddress>
 </unib:bookRent>
 </soapenv:Body>
</soapenv:Envelope>
```

## RentRequest

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
 instance">
 <SOAP-ENV:Body>
 <bookRentResponse>
 <departureDateTime xsi:type="xsd:string">?<departureDateTime>
```

```
<arrivalDateTime xsi:type="xsd:string">?<arrivalDateTime>
<status xsi:type="xsd:string">?</status>
<rentId xsi:type="xsd:string">?</rentId>
</bookRentResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

## Bindings

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

## Run service:

```
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. E' stato sviluppato utilizzando la libreria express.js, internamente utilizza le api di distancematrix.ai per il calcolo delle distanze.

## API:

| Risorsa       | Parametri                           |
|---------------|-------------------------------------|
| GET /distance | <b>from:</b> indirizzo di partenza. |
|               | <b>to:</b> indirizzo di arrivo.     |

## Esempio

```
http://localhost:8080/distance?from=Mura+Anteo+Zamboni+7+40126+Bologna+%28B0%29%0D%0A&to=Ferrara
```

## Come eseguire

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

## Eseguire con Docker Compose

```
docker-compose up
```

