

ACMESky

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

Si progetti e si realizzi una SOA che supporti le attività di ACMESky.

Workflow e artefatti Si modellino le comunicazioni dello scenario sopra esposto usando una coreografia, si discutano le sue proprietà di connectedness ed eventualmente si raffini la coreografia per migliorare tali proprietà. Si proietti la coreografia in un sistema di ruoli.

Utilizzando uno o più diagrammi di collaborazione BPMN si modelli l'intera realtà descritta compresi i dettagli di ogni partecipante riferibile ad ACMESky. Tale modellazione ha scopo documentativo quindi il livello di dettaglio deve essere consistente con tale scopo. I partecipanti "esterni" (compagnie aeree, sistema bancario, ecc...) possono apparire come collapsed pools.

Si progetti una SOA per la realizzazione del sistema e la si documenti utilizzando UML (eventualmente con opportuni profili, ad esempio TinySOA).

Le interfacce esposte dei servizi descritti nella modellazione (dove possono apparire in forma semplificata) dovranno poi essere effettivamente realizzate nell'implementazione.

Si realizzi il sistema usando come tecnologie un BPMS (si consiglia di utilizzare Camunda), Jolie e API Rest.

Il BPMS deve essere utilizzato per supportare i processi di ACMESky.

Si assume che il sistema integri sotto forma di servizi (almeno) le seguenti capability esterne:

Calcolo distanze geografiche (preferibilmente con API Rest) Sistema bancario Compagnie di noleggio con autista (Jolie) Compagnie aeree Prontogram (preferibilmente API Rest) Tali servizi vanno implementati (con logica elementare) come parte del progetto.

I modelli di processo BPMN da utilizzare per il BPMS devono essere consistenti con la modellazione a scopo documentativo precedentemente realizzata; volendo si può anche scegliere di dettagliare compiutamente già dal primo modello le pool eseguibili.

Il dialogo fra Jolie e BPMS deve avvenire via SOAP, si veda il sito del corso alla pagina delle risorse per informazioni ulteriori.

Consegna e discussione Gruppi: il progetto va realizzato in gruppi di 2/3 persone.

Tempi: Il progetto va consegnato prima che inizino le lezioni dell'A.A. 2021-22.

Materiale da consegnare: relazione che descrive il lavoro fatto nelle varie fasi di modellazione e sviluppo, inclusi i vari diagrammi prodotti: coreografia e sistema proiettato, diagramma/i UML, diagramma/i di processo BPMN, export del progetto del BPMS, sorgenti dei servizi Jolie ed eventuali sorgenti aggiuntivi.

Modalità di consegna: via email con allegati. Se la dimensione degli allegati fosse eccessiva si consiglia di utilizzare servizi quali wetransfer, dropbox o similari.

Discussione del progetto: la discussione avviene su richiesta. Alla discussione devono presenziare tutti i membri del gruppo. La valutazione è personale, il che vuol dire che i partecipanti di uno stesso gruppo possono ottenere voti differenti fra loro. Queste specifiche non possono considerarsi definitive e verranno corrette e/o integrate quando necessario.

Opzioni Fermo restando che la corretta realizzazione del progetto proposto senza la parte opzionale permette di ottenere comunque il massimo punteggio, viene proposta una consegna aggiuntiva da considerarsi opzionale:

Modellazione della coreografia anche attraverso un diagramma di coreografia BPMN. Revisioni Queste specifiche possono essere soggette a revisioni per chiarire eventuali ambiguità e integrare possibili mancanze.

V0.9 - versione interna RFC

| Service | Port |
|---------|------|
| | |

| | |
|------------------------|------|
| ACMEsky | - |
| web | 80 |
| api | 8080 |
| Postgres | 5050 |
| BankService | 8070 |
| AirlineNationalService | 8060 |
| AirlineService | 8061 |
| ProntogramService | 8050 |
| ProntogramFrontEnd | 8051 |
| GeographicalService | 8040 |
| RentService1 | 8030 |
| RentService2 | 8032 |
| RentService3 | 8034 |

Vincoli al problema (Assunzioni):

- Le offerte generate provengono dalla stessa compagnia aerea
- I dati del viaggiatore vengono presi dall'account
- Non viene gestito lo scambio di denaro dalla banca alla compagnia aerea

TODO Cancellazione logica del volo Rigenerare db Modificare CascadeType in model

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: 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: messaggio 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: messaggio 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 ;
      (
        // 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 -> AIRk ;
          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 alla compagnia di noleggio se distanza
<30Km
    1
    +
    // requestRent1: richiesta noleggio veicoli 1
    // responseRent1: risposta noleggio 1
    // requestRent2: richiesta noleggio veicoli 2
    // responseRent2: risposta noleggio 2
    (
        requestRent1: ACME -> RENTi1 ;
        responseRent1: RENTi1-> ACME ;
        requestRent2: ACME -> RENTi2 ;
        responseRent2: RENTi2-> ACME
    )
    )
    )
    )
    +
    (
        // Errore nel pagamento
        // unbookTickets: cancella la prenotazione dei biglietti
        // emitCoupon: pagamento fallito
        unbookTickets: ACME -> AIRk ;
        emitCoupon: ACME -> BANK
    )
    )
    // Errore nella prenotazione dei biglietti
    // errorTickets: errore volo non disponibile
    +
    errorTickets: ACME -> USERx
    )
    )
    +
    // ACMEsky controlla l'offerta e non è più disponibile
    // responseOfferError: errore offerta
    responseOfferError: ACME -> USERx
    )
)*

```

Verifica condizioni connectedness delle coreografie

Per stabilire la connectedness, e anche 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 (25).

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

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

```
4.   (
5.     (
6.       bookTickets: ACME -> AIRk ; 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 *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 (19)

```
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 + ( requestRent1: ACME -> RENTi1 ; responseRent1: RENTi1-> ACME ;
15.              requestRent2: ACME -> RENTi2 ; responseRent2: RENTi2-> ACME )
16.          )
17.      )
18.  )
```

E' connessa per la sequenza perché il destinatario di **requestRent1** è il mittente di **responseRent1**, il destinatario di **responseRent1** è il mittente di **requestRent2** e il destinatario di **requestRent2** è il mittente di **responseRent2**.

```
19.      ) + ( errorPaymentBank: BANK -> ACME ;
20.          unbookTickets: ACME -> AIRk ; unbookTicketsResponse: AIRk -> ACME ;
21.          emitCoupon: ACME -> BANK ; emitCouponResponse: BANK -> ACME
22.      )
```

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**.

```
22.      )
23.  ) + errorTickets: ACME -> USERx
```

E' connessa per la choice perchè i mittenti di (6) e di (23) sono gli stessi.

```
24.      )
25.  ) + responseOfferError: ACME -> USERx
26.  )
```

E' connessa per la choice i sender di (3) e di (25) sono gli stessi.

```
27. )*
```

La coreografia è connessa per l'iterazione in quanto (23) e (25) terminano con il ricevente **USER** che è il mittente di (1), mentre (22) 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 + (requestRent1@RENTi1 ; responseRent1@RENTi1 ;  
                    _____
```

```

requestRent2@RENTi,2 ; responseRent2@RENTi,2
))
)) _____
) + ( unbookTickets@AIRk ; emitCoupon@BANK )
)
_____
) + errorTickets@USERx
)
_____
) + responseOfferError@USERx
)
)*

```

Utente

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

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

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

$$\text{proj}(\text{NotificaOfferta}, \text{USER}_x) = \overline{(\text{1 ; notifyUser@PTG ; notifyResponse@PTG ; 1})^*}$$

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

```
proj(AcquistaOfferta, 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 )
)
) + errorTickets@ACME
)
) + responseOfferError@ACME
)
)*

```

Airline

```

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 ; responseTickets@ACME ;
                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
)
)*

```

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@USERx ;
      (
        ( _____
        successPaymentBank@ACME ;
        _____
        ( 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 + ( 1 ; 1 ; 1 ; 1 ) )
        ))
      ) + ( 1 ; 1 )
    )
  ) + 1
  )
) + 1
)
)*

```

Rent company

```
proj(QueryDeiVoli, RENTi1) =  
  ( 1 ; 1 )*
```

```
proj(RicezioneOfferteLastMinute, RENTi1) =  
  ( 1 ; 1 )*
```

```
proj(RegistrazioneInteresse, RENTi1) =  
  ( 1 ; 1 )*
```

```
proj(NotificaOfferta, RENTi1) =  
  ( 1 ; 1 ; 1 ; 1 )*
```

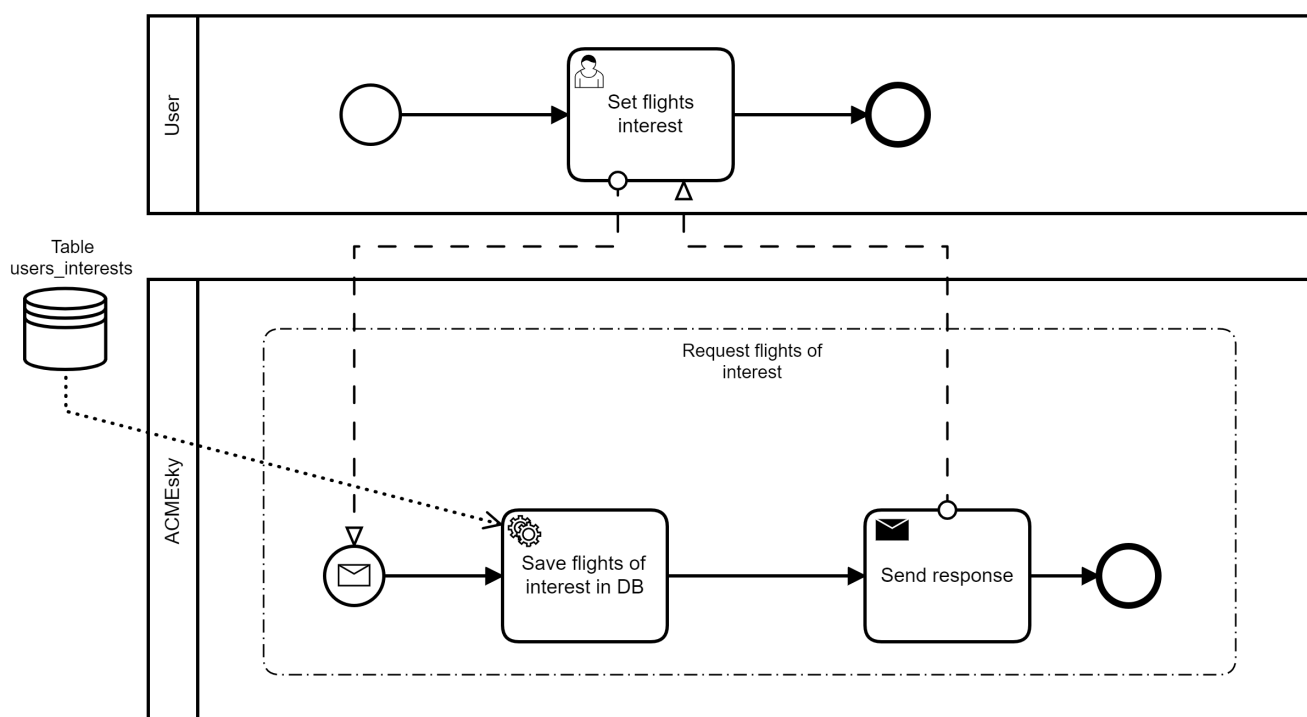
```
proj(RichiestaRicevuta, RENTi1) =  
  ( 1 ; 1 )*
```

```
proj(AcquistoOfferta, RENTi1) =  
  ( 1 ;  
    (  
      ( 1 ; 1 ;  
        (  
          ( 1 ; 1 ; 1 ; 1 ; 1 ; 1 ;  
            (  
              ( 1 ;  
                ( 1 + ( 1 ; 1 ;  
                  ( 1 + ( requestRent1@ACME ; responseRent1@ACME ;  
                    1 ; 1 ) )  
                ))  
              ) + ( 1 ; 1 )  
            )  
          ) + 1  
        )  
      ) + 1  
    )  
  )*
```


Diagramma BPMN


In questa sezione della documentazione si descrive il diagramma BPMN del progetto, in cui si mostra 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 offerte di volo, salvataggio di quelle last-minute, gestione delle offerte e pagamento, ecc.

Registrazione dell'interesse dell'utente



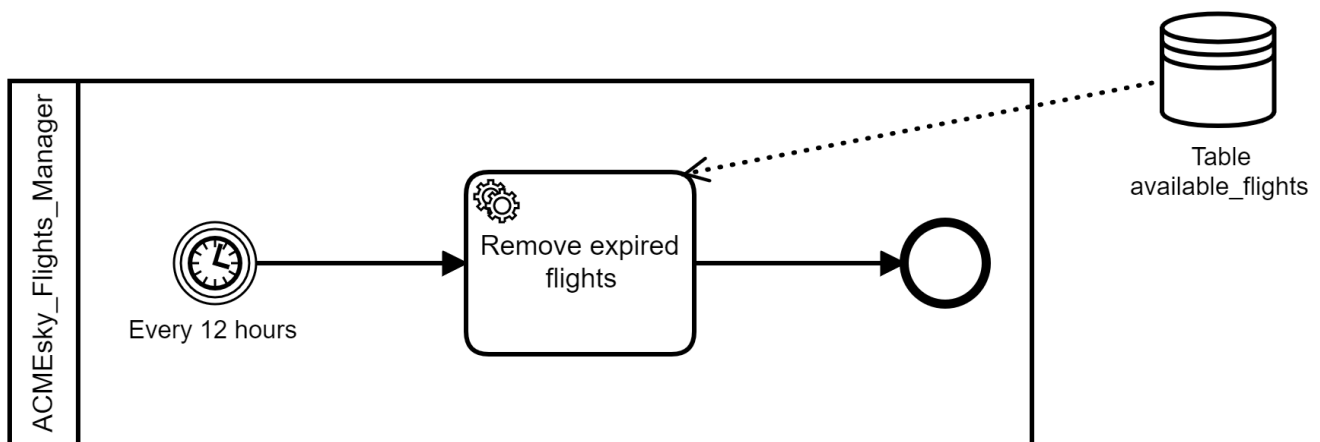
Il seguente diagramma descrive il processo di raccolta e registrazione dei voli di interesse degli utenti. Un utente descrive i suoi voli di interesse, specificando città/aeroporto di partenza, città/aeroporto di arrivo, data di partenza e ritorno del volo, e quota massima di spesa. Infatti le offerte di volo sono sempre formate dal volo di andata ed dal volo di ritorno. ACMEsky salva i voli di interesse nel suo Database, in particolare nella tabella "flights_interest" vengono salvati i voli che formano l'offerta di interesse e in "users_interests" l'offerta di interesse vera e propria. Infine invia la conferma di avvenuta creazione dei voli di interesse.

Registrazione delle offerte last-minute

 registrazione_voli_last-minute

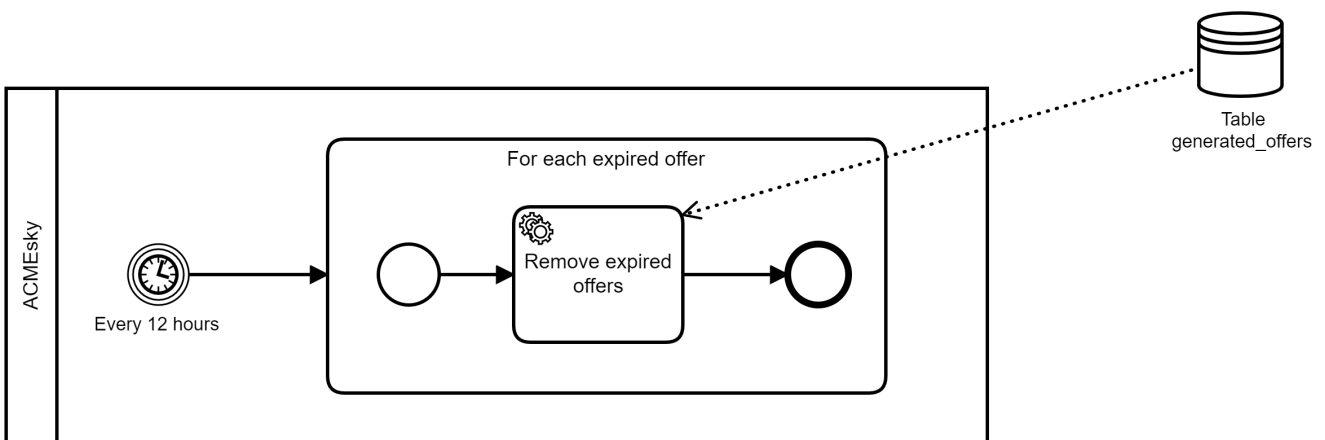
In questa parte si descrive il processo di ricezione e salvataggio dei voli last-minute. I servizi di Airline Service mandano voli last-minute appena creati ad ACMEsky, la quale salva ciascuno di essi nel database, nello specifico nella tabella available_flights.

Rimozione dei voli scaduti



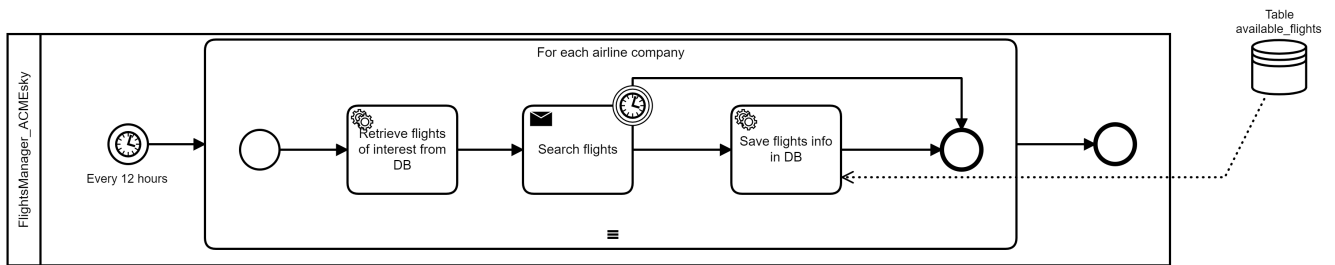
Il processo di cancellazione dei voli scaduti presenti nel Database avviene ogni 12 ore. I voli scaduti sono quei voli la cui data di scadenza è antecedente a quella in cui si effettua l'operazione di cancellazione. La cancellazione non comporta l'eliminazione effettiva del record che rappresenta quel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

Rimozione delle offerte scadute



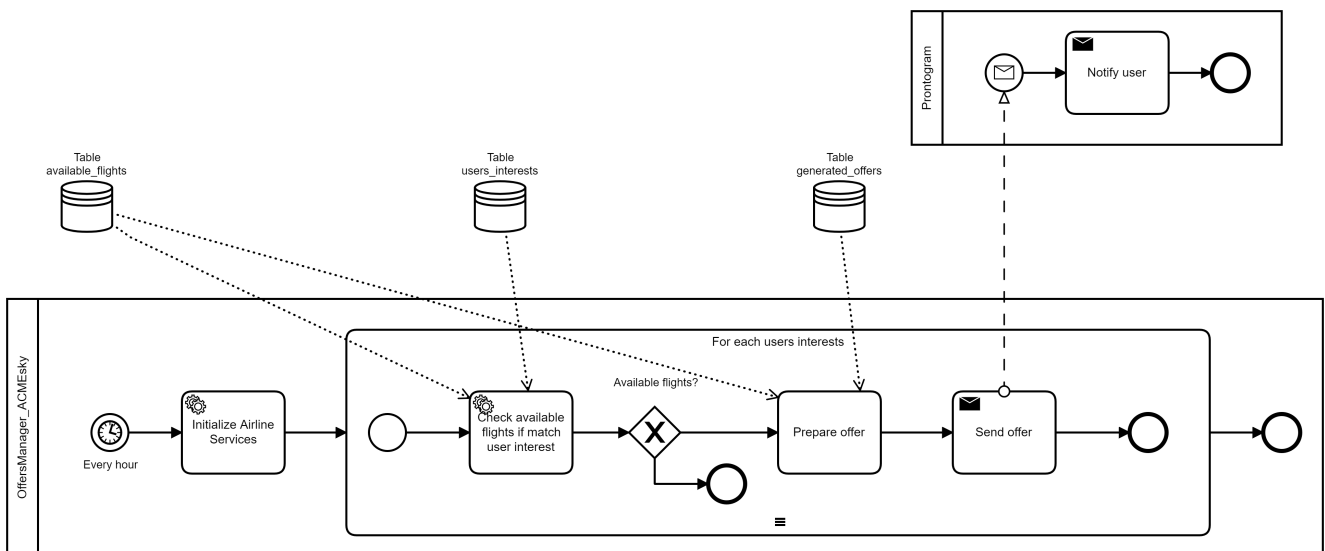
Il processo di cancellazione delle offerte di volo scadute presenti nel Database avviene ogni 12 ore. Le offerte di volo scadute comprendono i due voli di andata e ritorno che corrispondono ai voli di andata e ritorno di interesse degli utenti. Le offerte scadute sono quelle la cui data di scadenza del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

Registrazione voli



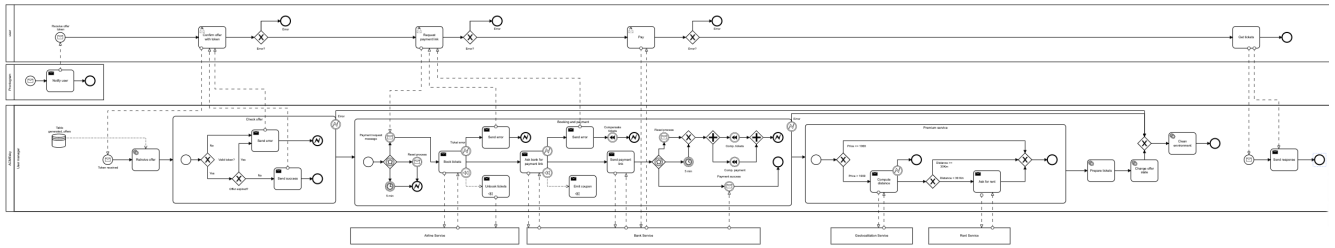
Ogni 12 ore e per ciascun Airline Service, ACMEsky recupera i voli di interesse degli utenti dal suo Database ed effettua una ricerca mirata dei voli compatibili con essi attraverso una chiamata ad una risorsa dei servizi di volo. I voli presenti nella risposta vengono salvati all'interno del DB (tabella available_flights). Se passa una certa quantità di tempo dovuto a ritardi o mancate risposte da parte dei Airline Service, per evitare che il flusso di esecuzione del sistema si fermi e non vada più avanti si è deciso di farlo terminare.

Match voli con interesse utente



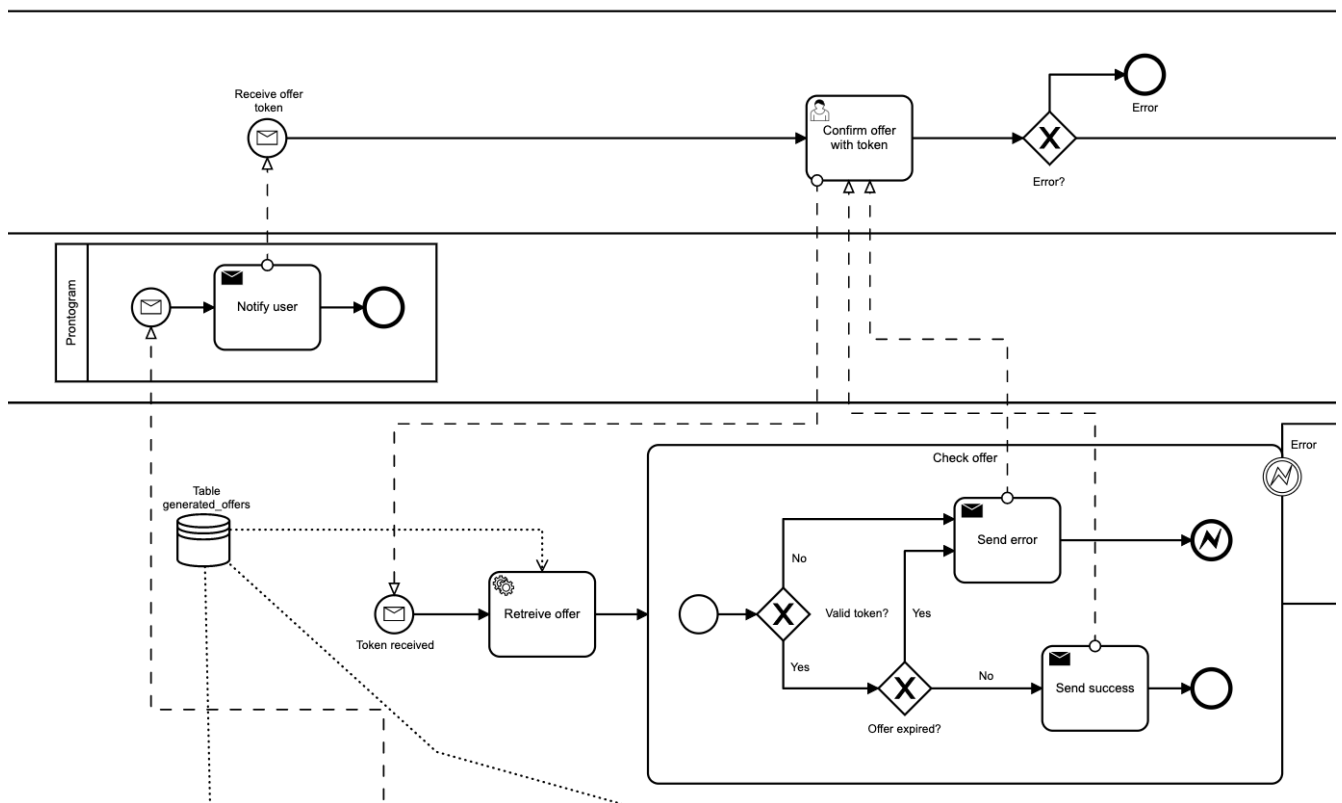
Ogni ora e per ciascun volo di interesse ACMEsky cerca tra i voli disponibili presenti nel DB (tabella available_flights) se c'è una corrispondenza con l'offerta di interesse presa in esame (offerta comprende voli di andata e ritorno di interesse). In caso negativo semplicemente il flusso termina, altrimenti, se si trovano i voli corrispondenti a quelli di andata e ritorno richiesti, si procede alla preparazione ed al salvataggio in DB dell'offerta comprensiva dei voli di andata e ritorno interesse che l'utente cercava (relazione generated_offers). Infine l'offerta di volo viene inviata all'utente attraverso l'app di Prontogram e il flusso termina.

Conferma di acquisto, applicazione servizi premium e preparazione biglietti



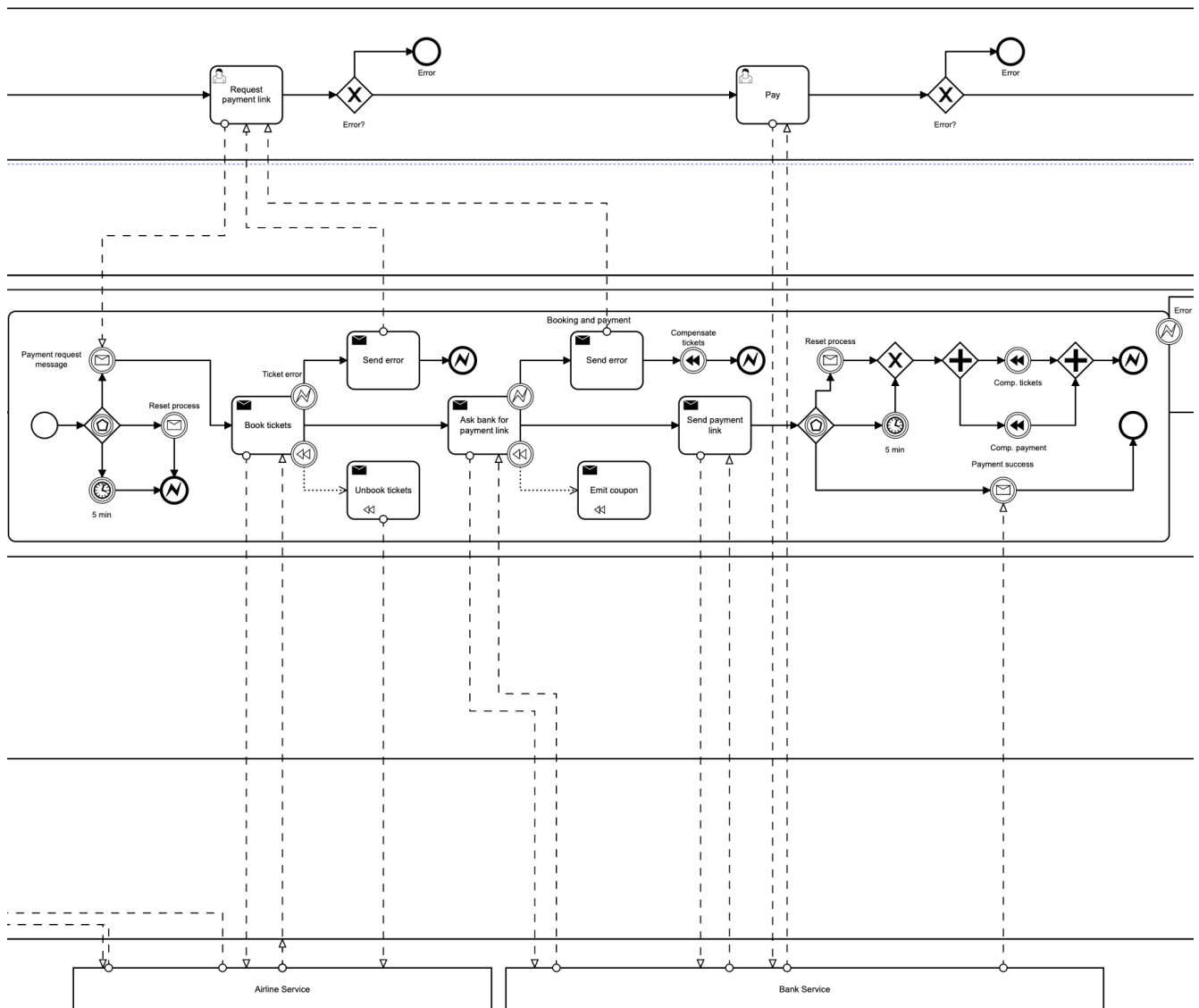
In questa parte del diagramma si illustra la conferma di acquisto dell'offerta di volo da parte dell'utente, l'acquisto dell'offerta e l'applicazione di servizi premium se l'offerta rispetta le caratteristiche del servizio premium. Infine vi è il controllo per verificare che non ci siano errori e si prepara il biglietto che poi il cliente avrà disponibile. Queste feature verranno suddivise in sotto parti del diagramma in modo da facilitarne l'esposizione.

Conferma dell'offerta da parte dell'utente



L'App di Prontogram notifica l'utente del fatto che c'è un'offerta i cui voli di partenza e ritorno corrispondono ai voli di interesse da lui richiesti e termina il suo flusso. 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 è ancora scaduta e se anche in questo caso l'esito è positivo si invia all'utente la conferma di accettazione dell'offerta, altrimenti lo si informa dell'esito negativo dovuto alla scadenza dell'offerta o del token ed il flusso termina con un errore.

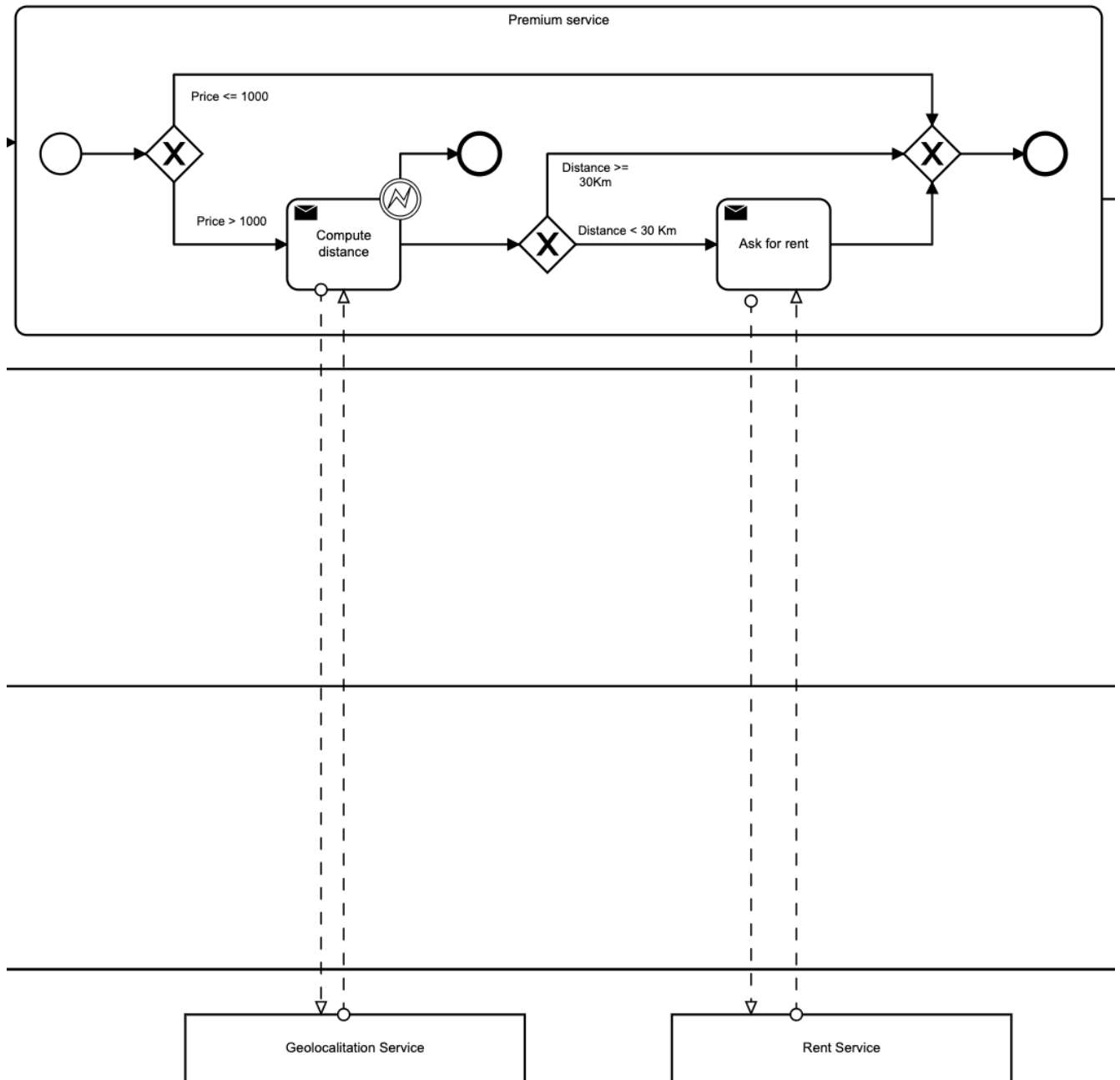
Pagamento dei voli



Il processo di pagamento inizia con una richiesta 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 offre 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 invece la prenotazione va a buon fine e ACMEsky dispone dei biglietti chiederà il link di pagamento a Bank Service, la quale glielo invierà in risposta appena possibile, ovvero la cancellazione della prenotazione. Il link viene sottoposto all'utente a meno di eventuali problemi che se presenti porterebbero alla compensazione dei biglietti precedentemente prenotati. L'utente sceglierà se pagare o meno. Se è intenzionato a pagare effettua il pagamento e riceverà risposta da Bank Service per comunicargli l'esito del pagamento. Bank Service comunica l'esito anche ad ACMEsky la quale potrà decidere in base ad esso di chiudere il flusso di pagamento oppure occuparsi delle compensazioni. Se il pagamento non è avvenuto con successo si effettua la compensazione del biglietto, altrimenti se il servizio della banca non risponde entro 5 minuti dal pagamento si procede alla compensazione dei biglietti e alla compensazione del pagamento, ovvero all'emissione di un coupon

spendibile dall'utente per acquistare nuove offerte di volo. In tal caso il flusso termina con un errore. Un errore nel pagamento (flusso ACMEsky) comporta il termine del flusso dell'utente.

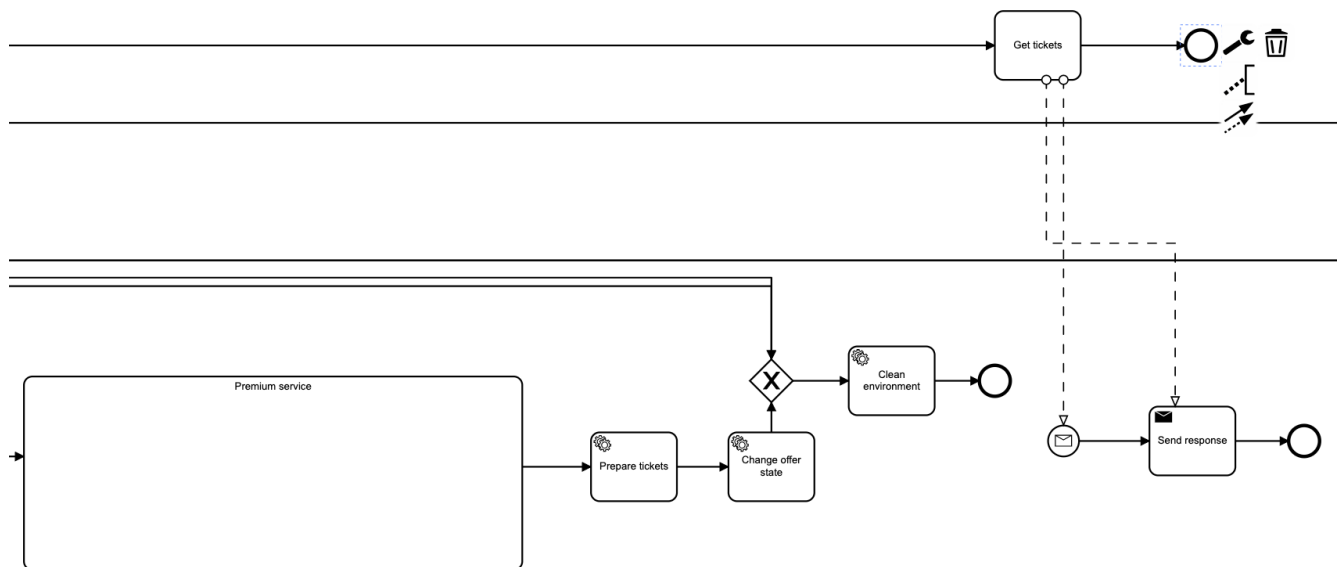
Servizi Premium voli



In questa fase ci si occupa di un eventuale applicazione del servizio premium all'offerta di volo dell'utente. All'inizio ACMEsky controlla il prezzo dell'offerta del cliente, poichè se questo supera i mille euro si invia una richiesta al servizio di geolocalizzazione per conoscere la posizione dell'utente. In seguito alla risposta del servizio, ACMEsky sarà in grado di calcolare la distanza tra il domicilio dell'utente e l'aeroporto di partenza. 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 gratuito dal suo domicilio all'aeroporto sia all'andata che al ritorno, e in

tal caso modificherà i biglietti includendo tutte le informazioni necessarie dei due trasferimenti. Chiaramente in caso la distanza sia inferiore il servizio di trasferimento gratuito non è disponibile. In caso il prezzo dell'offerta sia inferiore alla soglia chiaramente non verrà offerto nessun servizio premium. Infine il flusso termina.

Invio Biglietti



Arrivati a questo punto si cambierà lo stato dell'offerta riguardante l'acquisto e si prepareranno i biglietti che l'utente potrà scaricare. Il flusso termina per ACMEsky. L'utente può in qualunque momento richiedere i biglietti ad ACMEsky, la quale glieli restituirà. Il flusso termina sia per ACMEsky che per l'utente.

Diagramma delle coreografie BPMN

Diagrammi UML

In questa sezione della documentazione viene descritta la modellazione della "Service Oriented Architecture" (SOA) di cui fa parte l'organizzazione ACMEsky, sotto forma di diagrammi UML. Quest'ultimi sono stati implementati utilizzando il profilo TinySOA.

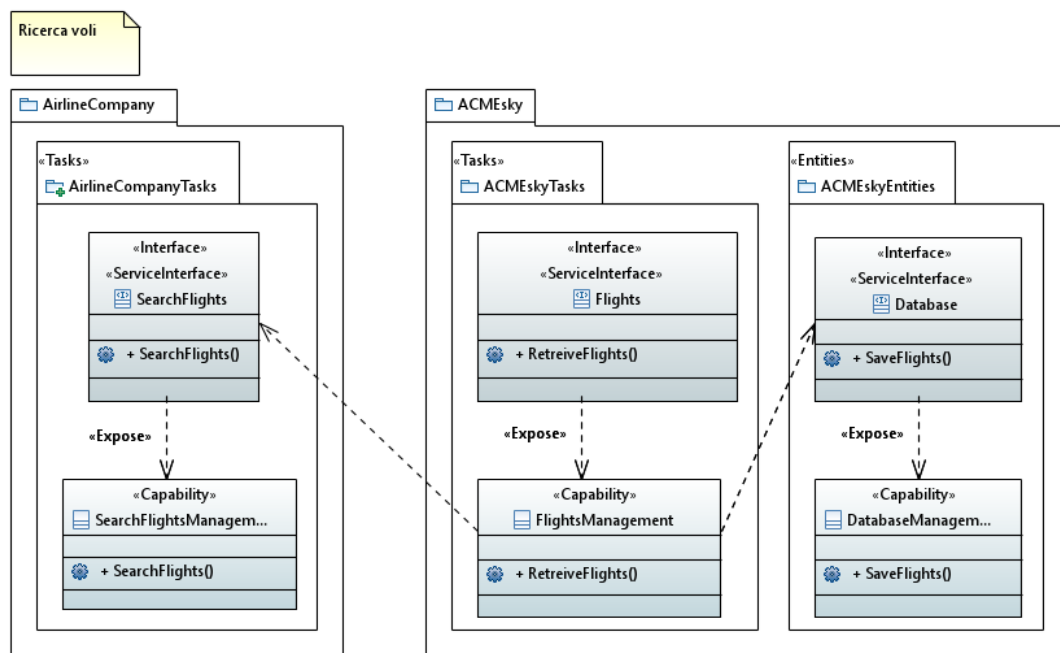
Lo scopo principale dei digrammi UML di seguito descritti, consiste nel riportare le interazioni che vi sono tra i vari servizi che fanno parte della SOA, attraverso l'esposizione di *capability* e interfacce che le espongono.

Nei diagrammi UML vengono riportati tre diverse tipologie di servizi:

- **Task:** espone le *capability* facenti parte dei processi interni all'organizzazione;

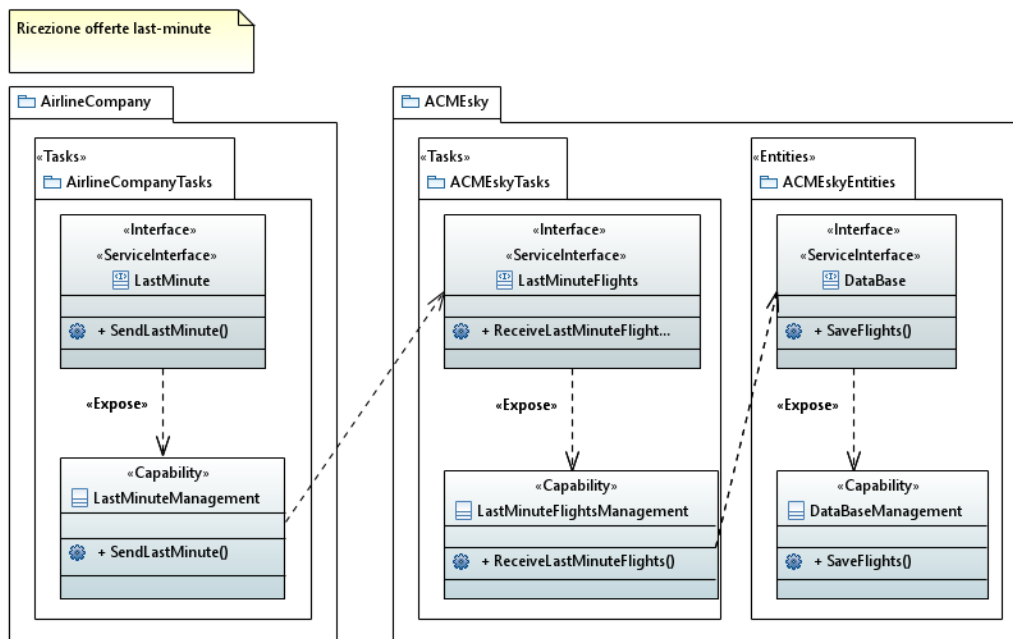
- **Entity:** fa riferimento ad una singola attività, possibilmente automatizzata (ad esempio: il salvataggio di un record all'interno di un Database);
- **Utility:** sono simili ai task, però non appartengono al dominio del problema (ad esempio: utilizzo di capability legate a servizi esterni).

Richiesta voli



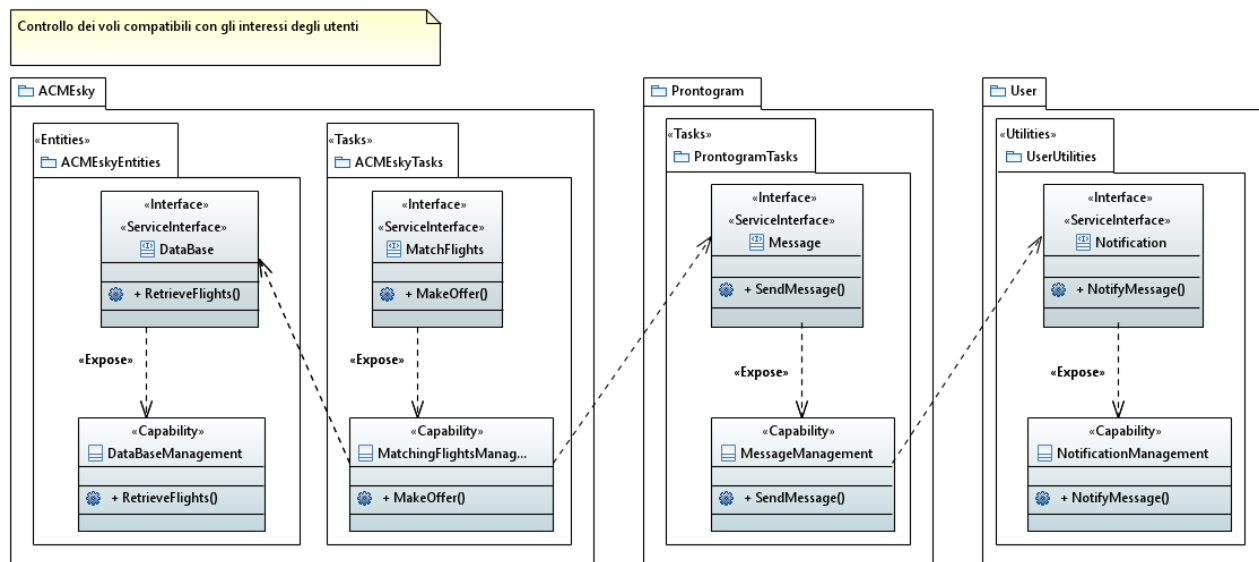
Nel diagramma riportato qui sopra vengono descritte le *capability* inerenti alla richiesta voli. In particolare per il ruolo di ACMEsky vengono espone le seguenti *capability*: *FlightsManagement* e *DatabaseManagement* le quali vengono espone da due interfacce *Flights* e *DataBase*. La *capability 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 di ACMEsky.

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 vengono espone 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*.

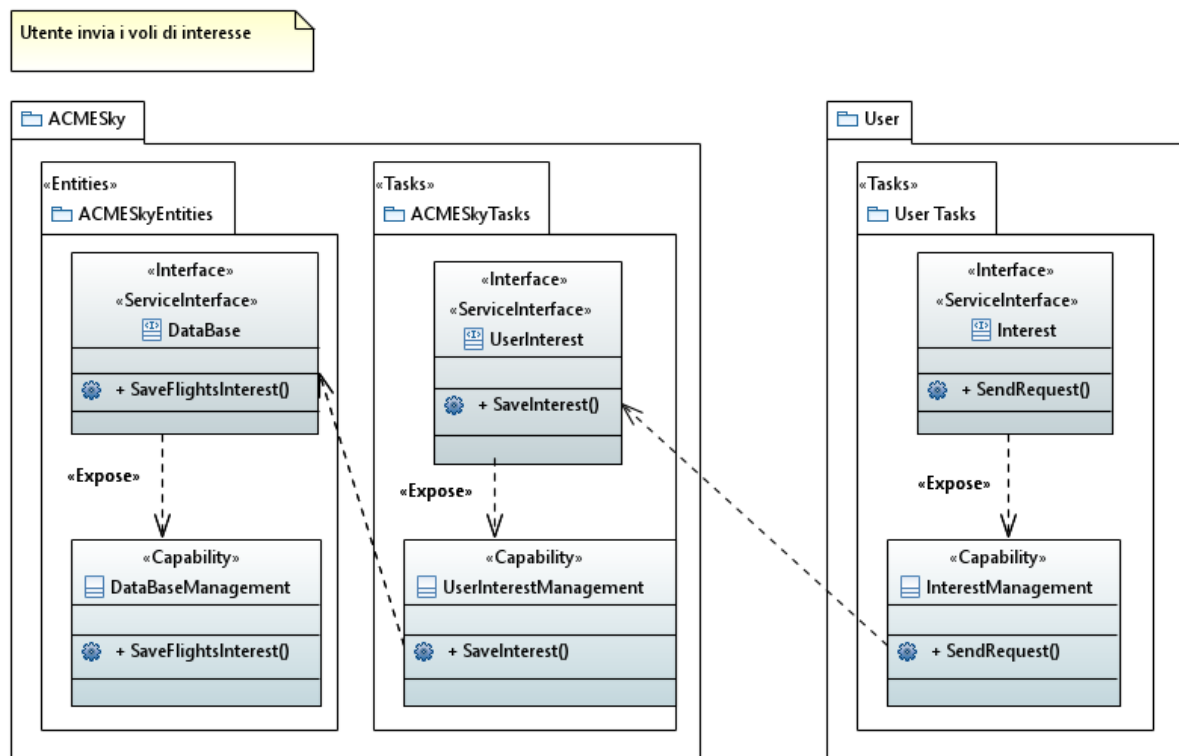
Controllo periodico dei voli compatibili con gli interessi degli utenti



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

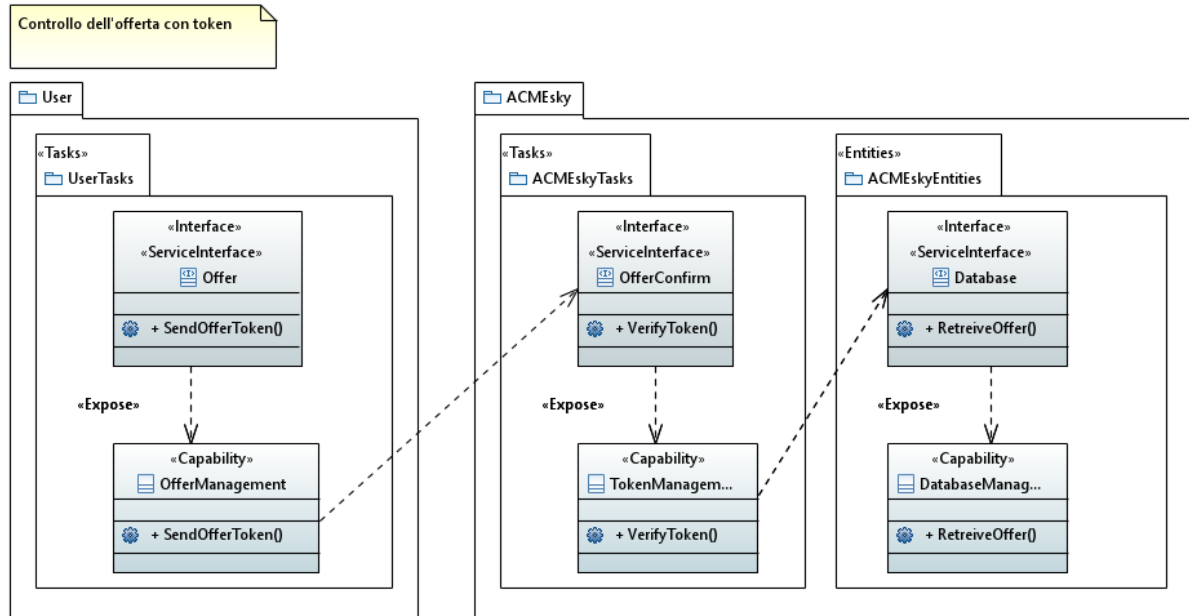
base dati attraverso *DatabaseManagement*, ha lo scopo di trovare l'offerta compatibile con quella di interesse per l'utente. Inoltre la capability *MatchingFlightsManagement* dipende dalla capability di invio messaggi di ProntoGram, *MessageManagement* esposta dalla relativa interfaccia: *Message*.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



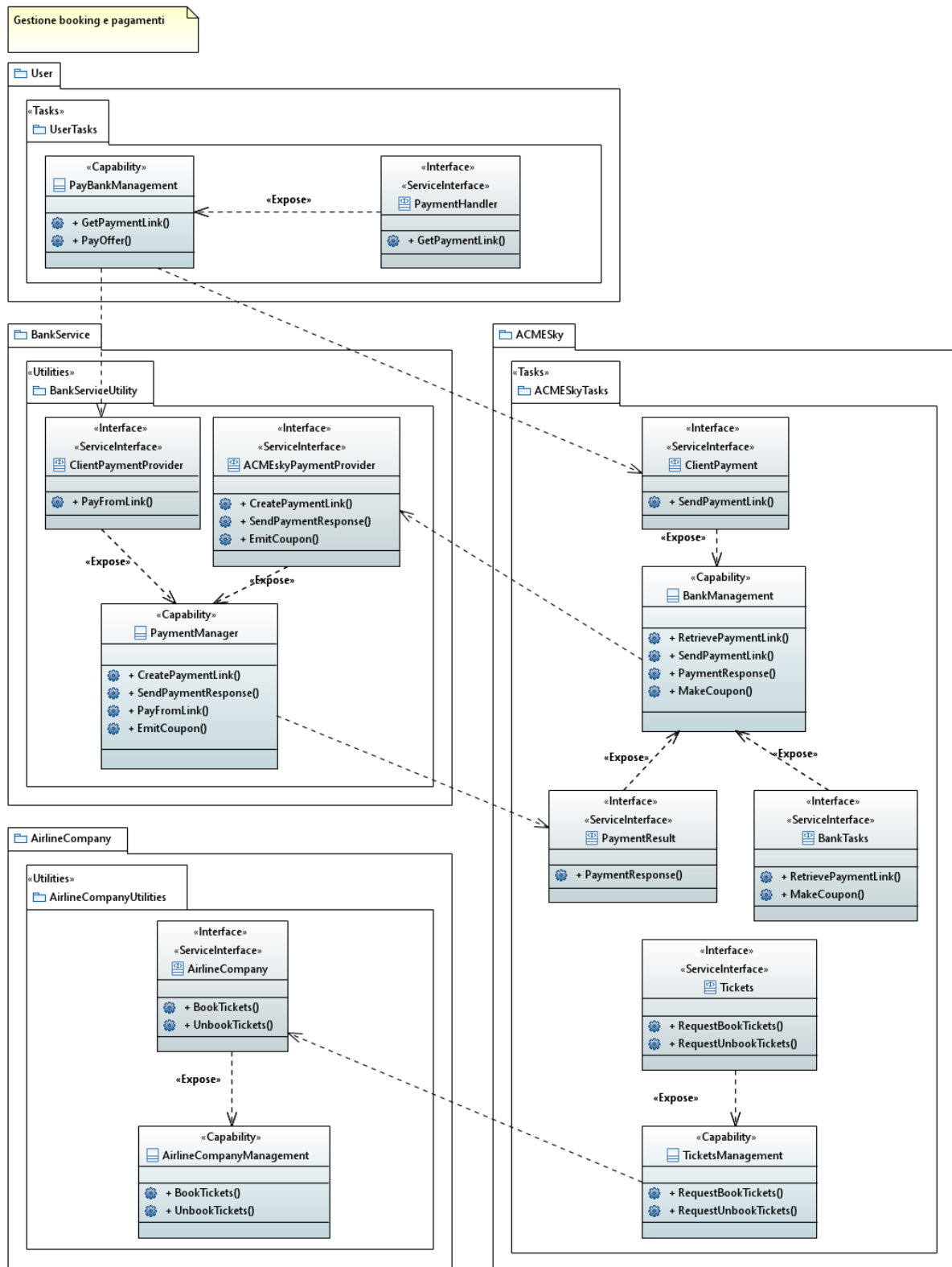
Nel diagramma riportato qui sopra vengono descritte le capability inerenti all'invio da parte di un utente della richiesta di un volo. In particolare per il ruolo di ACMESky vengono esposte le seguenti *capability*: *UserInterestManagement* e *DatabaseManagement* le quali vengono esposte da due interfacce 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 *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 vengono espone le seguenti *capability*: *TokenManagement* e *DatabaseManagement* le quali vengono espone 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.

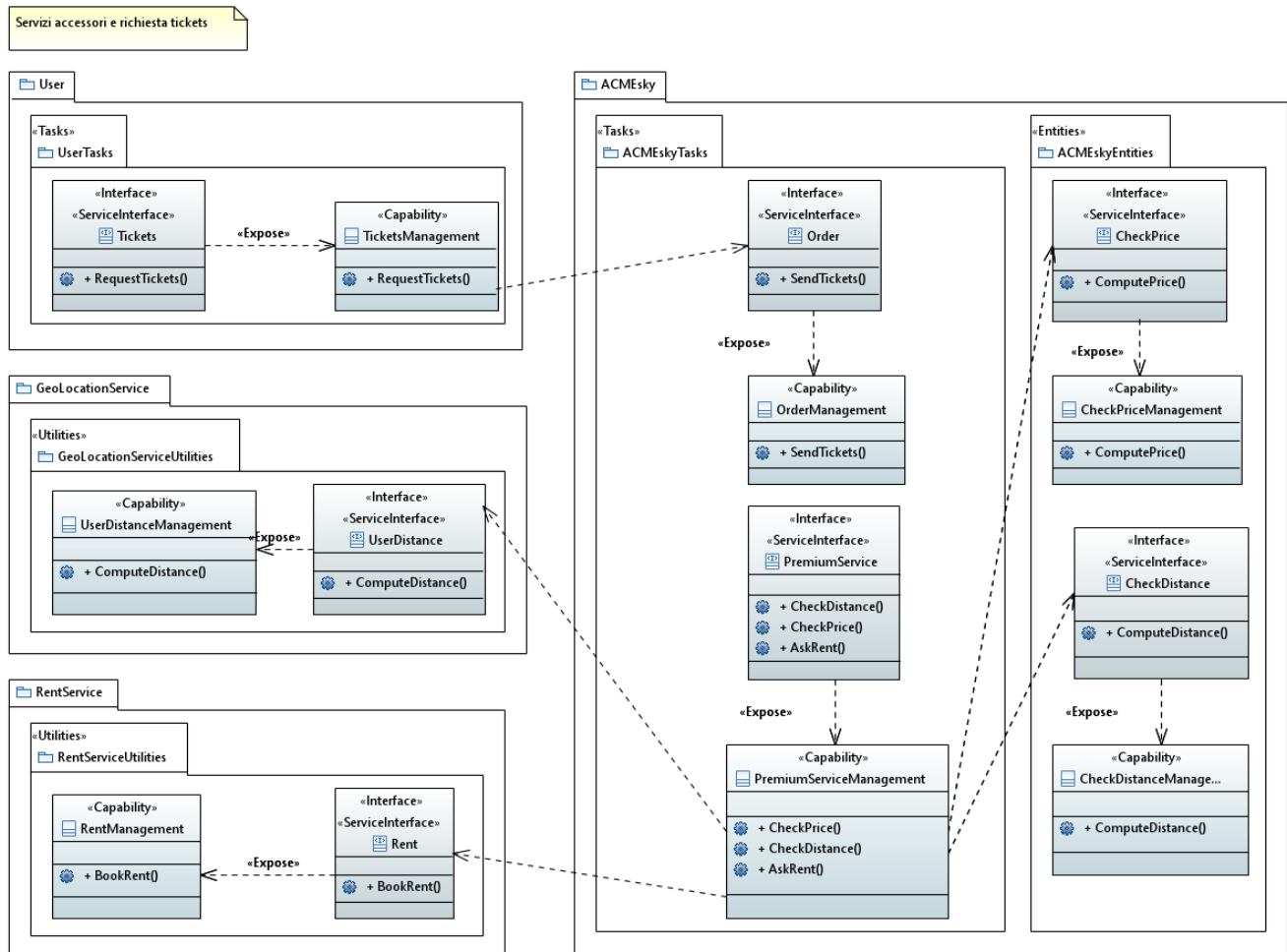
Gestione dei pagamenti



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 vengono espone le seguenti *capability*: *BankManagement* e *ArlineCompanyManagement* le quali vengono espone 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.

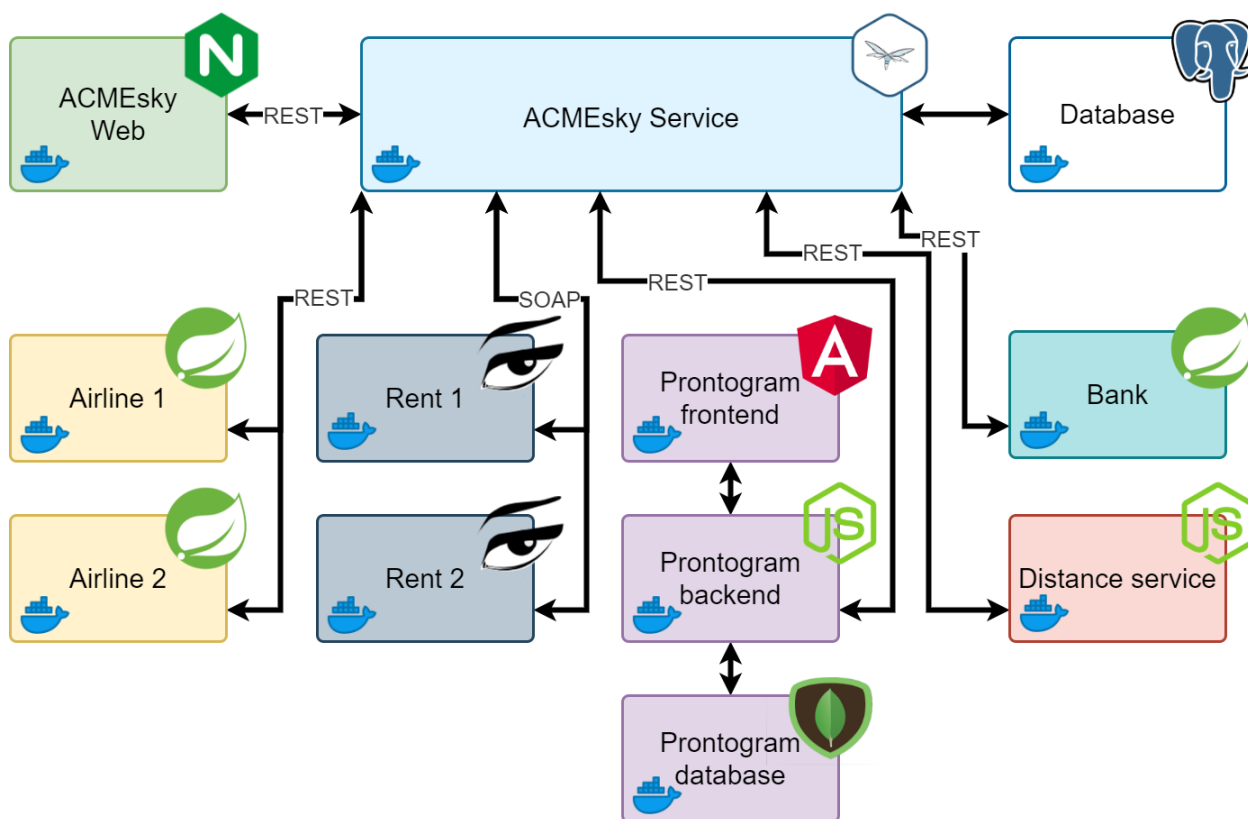
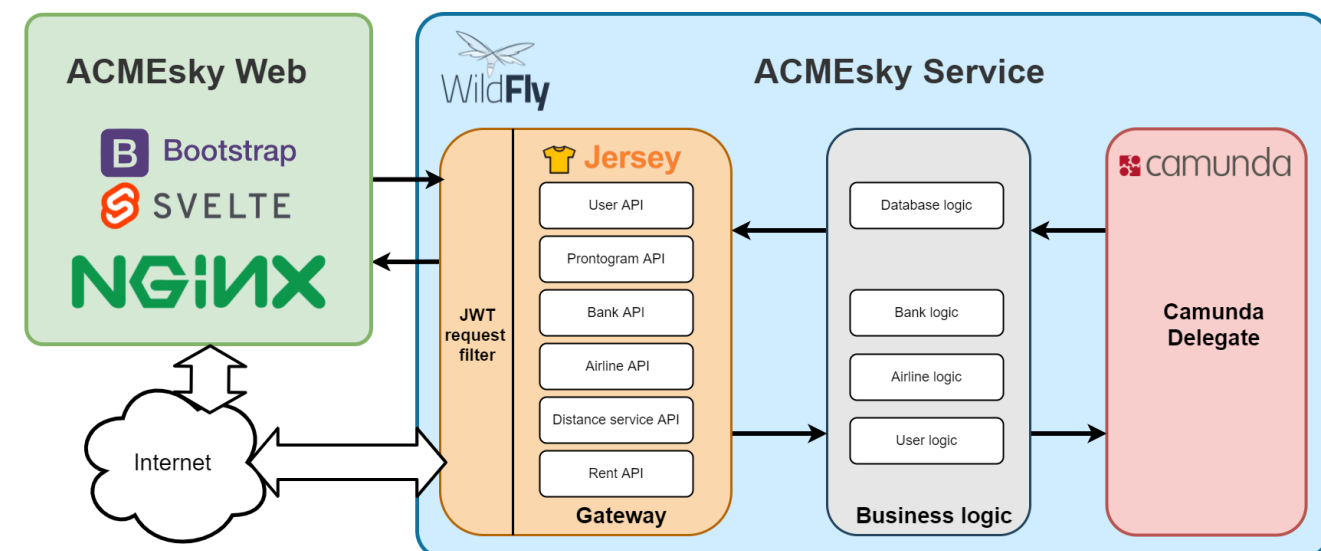
Servizi accessori e ricezione tickets



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

RentService(s) Come eseguire# 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
```

ACMEsky service

1.0 OAS3

<https://vallasc.github.io/ACMEsky/src/ACMEskyService/swagger/open-api.json>

ACMEskyService è il servizio principale di ACMEsky che si relaziona con gli altri (ACMEskyWeb, Prontogram, BankService,...) al fine di consentire agli utenti di prenotare ed acquistare le offerte e le richieste alle risorse che richiedono l'autenticazione i servizi devono autenticarsi effettuando una richiesta con username e password. Il token JWT ricevuto deve essere accompagnato ogni volta che viene fatta una richiesta a cui potrà accedere saranno differenti.

default

| | | |
|------|-----------------------|--------------------------------|
| POST | /airlines/last_minute | Riceve i voli last-minute |
| POST | /auth | Autenticazione |
| PUT | /auth/refresh | Refresh JWT |
| GET | /banks/confirmPayment | Conferma pagamento |
| GET | /airports/{code} | Get areoporto |
| GET | /airports | Lista degli areoporti filtrati |
| GET | /interests/{id} | Get interesse |

| | | |
|--------|-----------------|-------------------|
| GET | /interests/{id} | Get interesse |
| DELETE | /interests/{id} | Rimuove interesse |
| GET | /interests | Get interessi |
| | | |

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 aeroporti 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'aeroporto 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'aeroporto con un mezzo. Infine ACMEsky aggiunge i dettagli precedentemente descritti.

Panoramica

Struttura del servizio e tecnologie utilizzate

tecnologie e struttura

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'aeroporto per recuperare la lista dei suggerimenti per gli aeroporti 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'aeroporto per recuperare l'aeroporto di partenza e di arrivo dei voli sulla base dei codici aeroportuali 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 UserSignUpDTO, 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 aeroporti, 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 aeroporti nazionali e internazionali codificati secondo il codice IATA. Essa contiene il campo "id" (Chiave primaria), "code", ossia il codice dell'aeroporto in codifica IATA, "name", il nome dell'aeroporto, "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'aeroporto.

| | 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'aeroporto 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'aeroporto di partenza del volo con il quale è registrato nella tabella "airports" (chiave esterna), "arrival_airport_id" è l'identificativo dell'aeroporto 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 flag 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'aeroporto. 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.

Looking for a shareable component template? Go here --> [sveltejs/component-template](https://github.com/sveltejs/component-template)

svelte app

This is a project template for [Svelte](https://github.com/sveltejs/svelte) apps. It lives at <https://github.com/sveltejs/template>.

To create a new project based on this template using [degit](#):

```
npx degit sveltejs/template svelte-app
cd svelte-app
```

Note that you will need to have [Node.js](#) installed.

Get started

Install the dependencies...

```
cd svelte-app
npm install
```

...then start [Rollup](#):

```
npm run dev
```

Navigate to localhost:5000. You should see your app running. Edit a component file in `src`, save it, and reload the page to see your changes.

By default, the server will only respond to requests from localhost. To allow connections from other computers, edit the `sirv` commands in `package.json` to include the option `--host 0.0.0.0`.

If you're using [Visual Studio Code](#) we recommend installing the official extension [Svelte for VS Code](#). If you are using other editors you may need to install a plugin in order to get syntax highlighting and intellisense.

Building and running in production mode

To create an optimised version of the app:

```
npm run build
```

You can run the newly built app with `npm run start`. This uses [sirv](#), which is included in your `package.json`'s `dependencies` so that the app will work when you deploy to platforms like [Heroku](#).

Single-page app mode

By default, `sirv` will only respond to requests that match files in `public`. This is to maximise compatibility with static filesystems, allowing you to deploy your app anywhere.

If you're building a single-page app (SPA) with multiple routes, `sirv` needs to be able to respond to requests for *any* path. You can make it so by editing the `"start"` command in `package.json`:

```
"start": "sirv public --single"
```

Using TypeScript

This template comes with a script to set up a TypeScript development environment, you can run it immediately after cloning the template with:

```
node scripts/setupTypeScript.js
```

Or remove the script via:

```
rm scripts/setupTypeScript.js
```

Deploying to the web

With [Vercel](#)

Install `vercel` if you haven't already:

```
npm install -g vercel
```

Then, from within your project folder:

```
cd public
vercel deploy --name my-project
```

With [surge](#)

Install `surge` if you haven't already:

```
npm install -g surge
```

Then, from within your project folder:

```
npm run build
surge public my-project.surge.sh
```

AirlineService

A demo version of AirlineService

Build fat Jar:

```
mvn package
```

Create and run on Docker-compose container

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

API:

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

AirlineService API 1.0.5 OAS3

<https://vallasc.github.io/ACMEsky/src/AirlineService/swagger.json>

A description about all callable AirlineService API.

Servers

<http://localhost:8060> ▼

flight-offer-controller

POST /getFlights

GET /getTickets

GET /getIban

GET /getAll

POST /notPurchasedOffer

Models

UserRequest >

Flight >

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:
```

Documentation

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 raccolti da ACMEsky stessa. I voli vengono generati non appena il servizio viene attivato e inviati ad ACMEsky su richiesta dei clienti, mentre i voli last-minute vengono generati e passati ogni 10 minuti indipendentemente dalla presenza di voli di interesse ad essi corrispondenti. Il servizio manda i biglietti relativi ai voli per cui gli utenti manifestano la volontà di

acquistare tramite il servizio di ACMEskyWeb e infine ne cambia lo stato di disponibilità. 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 aeroporti nazionali, mentre la seconda offre voli da aeroporti 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. È 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 aeroporti 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 contenente 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

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 aeroporti 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. |

Bank service

Descrizione

API

[API link](#)

Bank service v1 OAS3

<https://vallasc.github.io/ACMEsky/src/BankService/openapi.json>

Il token jwt di default è precaricato

Servers

<http://localhost:8070> ▼

user-controller

POST /users

GET /users/me

payment-controller

GET /payments

POST /payments

auth-controller

POST /auth

Models

UserRequest >

How to build

```
mvn package
```

How to run

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

Run on a Docker container

```
docker-compose up
```

Database console

Go to <http://localhost:8080/h2>

Credentials:

URL: jdbc:h2:./db/bankdb

user: sa

passw:

ProntoGram

Prontogram's web-application

How to build

How to run

docker

How to build

API

[API link](#)

Prontogram API 1.0.0

[Base URL: prontogram.swagger.io/v2]
<https://vallasc.github.io/ACMEsky/src/Prontogram/api.json>

These are the Prontogram's API, they manage everything about users, notifications and subscrip

[Terms of service](#)

[Contact the developer](#)

[Contact the developer](#)
[Apache 2.0](#)
[Find out more about Swagger](#)

Schemes

HTTPS



user Operations about users

POST `/user` Add a new user to the DB

PUT `/user/{id}` Update an existing user

DELETE `/user/{id}` Deletes a user by ID

GET `/user/{id}` Find user by ID

GET `/user/all` Finds all users in the DB

notification Operations about notification and subscription

POST `/notification` Creates a new notification to the DB

DELETE `/notification/{notificationId}` Deletes a notification by Id

PUT `/notification/{notificationId}` Update an existing notification

GET `/notification/{notificationId}` Find notification by ID

GET `/notification/all/{username}` Find all notification by user ID

authentication Operations about registartion and login

GET **/auth/register** Returns the new user registered in the system

GET **/auth/login** Logs user into the system

subscription Operations about subscription

GET **/subscription/all** Finds all subscriptions in the DB

POST **/subscription/new** Creates a new subscription to the DB

DELETE **/subscription/{subendpoint}** Deletes a user's subscription by sub

Models

Notification >

User >

Subscription >

Documentation

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 front-end si occupa di creare l'interfaccia grafica e gestire le iterazioni da parte dell'utente.

Panoramica

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 multiplatforma 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.

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

Rent Service

Run:

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

Create and run the docker stack

```
docker-compose up
```

Geographical Distance Service

Run:

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

Create and run on Docker container

```
docker build -t distance_service .  
docker run --rm -d -p 8082:8080/tcp distance_service
```

API:

GET /distance?from=bologna&to=ferrara