# Università degli studi di Milano-Bicocca Corso di Ingegneria del software A. A. 2019/2020

Relazione sul progetto “Applicativo di gestione dei biglietti di compagnia aerea”

# A cura di: Anelli Pietro (matricola 830684) Brandalese Luca (matricola 829818) Condello Paolo (matricola 829800) Conti Simone (matricola 829621)

# Indice

1. Introduzione – pag. 3
2. Diagramma dei casi d’uso – pag. 4  
   2.1 Requisiti funzionali – pag. 7  
   2.2 Requisiti di sistema – pag. 8
3. Diagramma delle classi a livello di dominio – pag. 9
4. Diagramma delle classi a livello di progettazione – pag. 11
5. Diagramma di sequenza – pag. 15
6. Diagramma dell’architettura software – pag. 16
7. Diagramma degli stati – pag. 17
8. Diagramma di attività – pag. 18
9. Pattern utilizzati (MVC, DTO, Repository) – pag. 19

# Introduzione

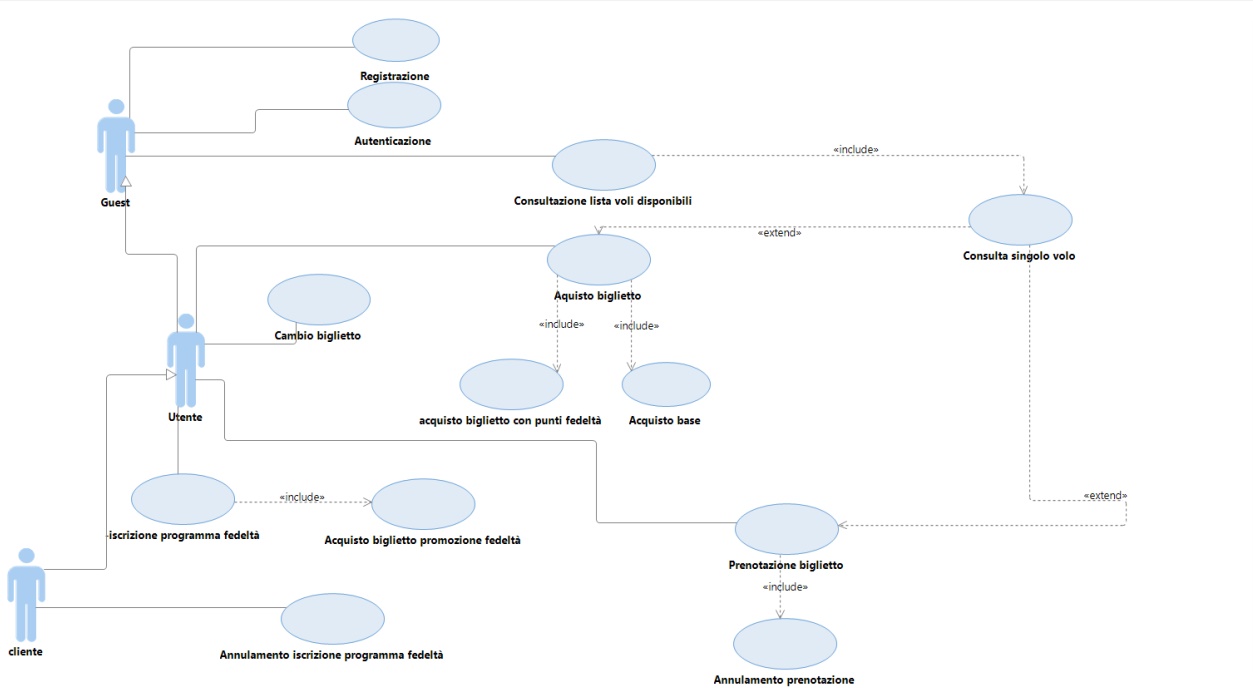
Questo progetto consiste nella realizzazione di un applicativo web che sia di supporto per delle compagnie aeree. L’applicativo consentirà all’utilizzatore di mettere in vendita online dei biglietti aerei, e gestire voli e clienti della compagnia aerea. La lista completa dei requisiti funzionali si trova a pagina 7.

Abbiamo deciso di applicare tre pattern per la stesura dell’applicativo: **MVC, DTO e Repository**, spiegheremo le motivazioni e il loro funzionamento a pagina

Questo applicativo è stato creato con l’utilizzo dei seguenti software:

* Eclipse IDE for Enterprise Java Developers – per la stesura del codice
* Apache Tomcat 9.0 – per i server
* Understand – per verificare la presenza di eventuali anti-pattern architetturali
* SonarCloud – per testare la qualità del codice
* IBM Rational Software Architect Designer – per la stesura dei diagrammi

# Diagramma dei casi d’uso



Nel diagramma di caso d’uso abbiamo tre attori: **Guest, Utente e Cliente.**Il **Guest** è un **utente non loggato**, **L’Utente** è un utente **loggato ma non iscritto al programma fedeltà**, mentre il **Cliente** è un utente **loggato e iscritto al programma fedeltà**.

Sono elencati di seguito tutti i dettagli dei casi d’uso:

|  |  |  |
| --- | --- | --- |
| **CASO D'USO** | **ATTORI COINVOLTI** | **SCENARI PRINCIPALI E ALTERNATIVI** |
| **Registrazione** | Guest | Il guest si registra al sito della compagnia aerea.  **SCENARIO PRINCIPALE:** Dopo avere inserito tutti i dati richiesti la registrazione va a buon fine, e il sistema inserisce nel database i dati dell'utente in questione. **SCENARIO ALTERNATIVO:** Se l'utente non inserisce tutti i dati richiesti, oppure inserisce dei dati errati, il sistema non confermerà la sua registrazione e inviterà l'utente a ricompilare il modulo di registrazione. |
| **Autenticazione** | Guest | Il guest potrà effettuare il login al sito. **SCENARIO PRINCIPALE:** Se i dati da lui immessi coincidono con quelli presenti nel database, allora effettuerà l'accesso al sito. **SCENARIO ALTERNATIVO:** Se i dati da lui immessi non coincidono con quelli presenti nel database, il login non andrà a buon fine e verrà chiesto all'utente di ritentare la fase di login. |
| **Consultazione lista voli disponibili** | Guest, Utente, Cliente | L'utente può consultare la lista dei voli disponibili inserendo almeno uno tra questi tre campi: data(GG/MM/AA), aeroporto di partenza e aeroporto di arrivo, e inserendo obbligatoriamente il numero di biglietti che vuole acquistare. Nella lista verranno visualizzati solo i voli che hanno abbastanza posti disponibili. |
| **Consulta singolo volo**  (include da consultazione lista voli disponibili) | Guest, Utente, Cliente | Una volta consultata la lista di biglietti disponibili (vedi caso d’uso consultazione lista voli disponibili) un qualsiasi utente può visualizzare i dettagli di un singolo biglietto, questo è anche l’unico modo per procedere all’acquisto. |
| **Acquisto biglietto**  (extend da Consulta singolo volo) | Utente, Cliente | L'utente una volta selezionato un volo dalla lista dei voli disponibili (vedere caso d'uso consulta volo singolo) potrà decidere di acquistare i biglietti da lui richiesti. L’utente dovrà prima selezionare se effettuare l’acquisto usando punti fedeltà oppure eseguire un acquisto classico usando la carta di credito associata al suo account. **SCENARIO PRINCIPALE:** L'utente riesce ad acquistare i biglietti, di conseguenza i numeri di posti disponibili per il volo caleranno del numero di biglietti acquistati dall’utente, e gli verranno attribuiti i biglietti in questione. Inoltre il sistema notificherà l'utente dell'avvenuto pagamento.  **SCENARIO ALTERNATIVO:** L'utente per effettuare l'acquisto dovrà essere loggato. In caso non lo fosse il sistema lo reindirizzerà al login. |
| **Acquisto biglietto promozione fedeltà**  (include da Iscrizione programma fedeltà) | Cliente | Ogni cliente può acquistare biglietti a prezzi scontati.  **SCENARIO PRINCIPALE:** Il sistema verifica che l'utente faccia parte del programma fedeltà, e in tal caso applica lo sconto sui biglietti. **SCENARIO ALTERNATIVO:** Il sistema verifica che l'utente non fa parte del programma fedeltà e di conseguenza impedisce l'applicazione della promozione sull'acquisto dei biglietti |
| **Cambio biglietto** | Utente, Cliente | L’utente, pagando la differenza tra i due biglietti, può cambiare la data di uno o più dei suoi biglietti da lui precedentemente acquistati (al massimo 3 giorni prima della partenza), con stesso aeroporto di partenza e arrivo, entro la data di partenza del volo per un altro biglietto dello stesso volo ma in data differente. **SCENARIO PRINCIPALE:** Le prenotazioni precedenti vengono cancellate e sostituite da quelle appena selezionate dall'utente, che riceverà una notifica di avvenuta sostituzione dei biglietti. |
| **Prenotazione biglietto**  (include da Consulta volo singolo) | Utente, Cliente | Questo caso d'uso funziona in modo analogo al caso d'uso "acquisto biglietto", con la differenza che dopo aver selezionato il numero dei biglietti ed il volo, l’utente non sarà obbligato ad eseguire il pagamento immediato.  **SCENARIO PRINCIPALE:** Al volo in questione verranno sottratti tanti posti quanti i biglietti prenotati dall'utente. Nel database n biglietti del volo saranno associati all'utente. Una volta avvenuto tutto questo, il cliente verrà notificato dal sistema dell'avvenuta prenotazione. **SCENARIO ALTERNATIVO:** Se il cliente non è loggato, allora il sistema impedirà di proseguire con la prenotazione e reindirizzerà l'utente al login. |
| **Annullamento prenotazione**  (extend da Prenotazione biglietto) | Utente, Cliente | L'utente potrà decidere di annullare una o più prenotazioni a suo carico, e nel farlo dovrà specificare quanti biglietti rimuovere dalla prenotazione in questione. Tutto questo può essere effettuato solo prima di 3 giorni dalla partenza del volo in questione. |
| **Iscrizione programma fedeltà** | Utente | In un qualsiasi momento l'utente può decidere di iscriversi al programma fedeltà. Una volta effettuata la registrazione al programma fedeltà, l'utente in questione verrà classificato come cliente, e nei suoi dati all'interno del database verrà aggiunto un codice univoco "fedeltà". L'iscrizione al programma fedeltà potrà essere eseguita solo da un utente loggato. |
| **Acquisto biglietto con punti fedeltà**  (include da Acquisto biglietto) | Cliente | Questo caso d'uso funziona in modo analogo al caso d'uso "acquisto biglietto", con la differenza che al posto di usare dei soldi reali, il cliente utilizza i punti fedeltà, se ne ha a sufficienza per coprire il prezzo (in punti fedeltà) del biglietto. |
| **Acquisto base**  (include da Acquisto biglietto) | Utente, Cliente | Questo caso d’uso consente all’utente di acquistare un biglietto con il metodo classico attraverso carta di credito. Per scenari principali e alternativi vedere il caso d’uso Acquisto biglietto. |
| **Annullamento iscrizione programma fedeltà** | Cliente | Se un cliente non acquista nessun biglietto nell'arco di due anni, verrà avvisato via e-mail dal sistema che un ulteriore anno senza acquistare alcun biglietto comporterebbe l'annullamento del programma fedeltà a lui riservato. Anche l'annullamento viene effettuato in modo automatico dal sistema. |

## 2.1 Requisiti funzionali

I requisiti funzionali sono tutte quelle azioni che il nostro applicativo offre. Anche solo togliendo uno di questi requisiti l’applicativo non funzionerebbe completamente.

Ciò significa che nei requisiti funzionali troveremo tutti i casi d’uso elencati (spiegati nel dettaglio nella tabella sopra):

* **Registrazione**
* **Autenticazione**
* **Consultazione lista voli disponibili**
* **Consulta singolo volo**
* **Acquisto biglietto**
* **Acquisto biglietto con punti fedeltà**
* **Acquisto base**
* **Acquisto biglietto promozione fedeltà**
* **Cambio biglietto**
* **Prenotazione biglietto**
* **Annullamento prenotazione**
* **Iscrizione programma fedeltà**
* **Acquisto biglietto promozione fedeltà**
* **Annullamento iscrizione programma fedeltà**

Inoltre nei requisiti funzionali andranno aggiunte le seguenti funzionalità, che però non possono essere rappresentate come caso d’uso in quanto eseguite automaticamente dal sistema:

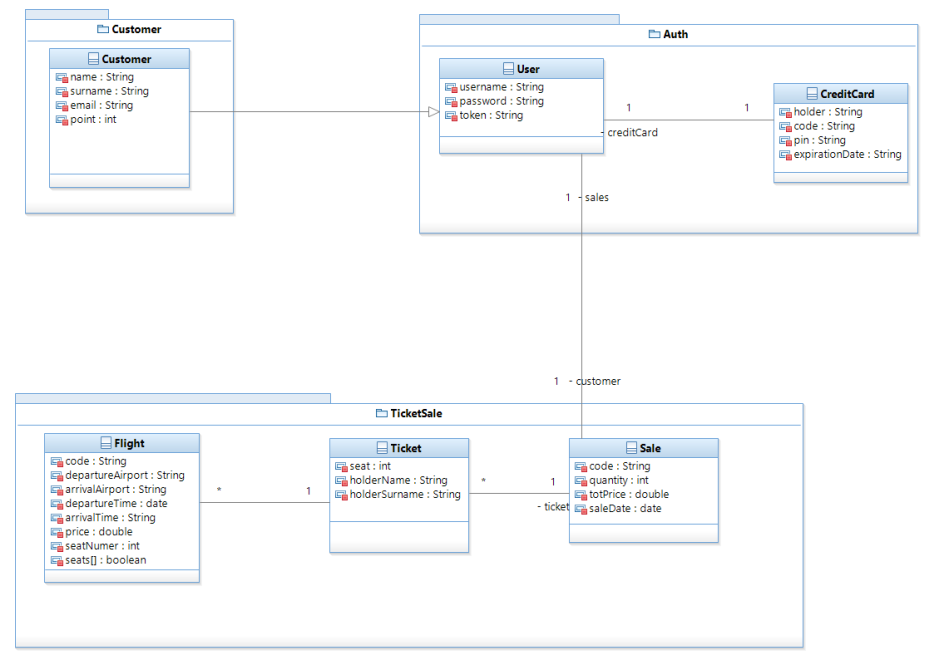
* **Cancellazione prenotazione:** Una prenotazione viene automaticamente cancellata dal sistema se, entro tre giorni dalla partenza del volo, il cliente non ha ancora effettuato il pagamento.
* **Accumulo punti fedeltà:** Per ogni cliente iscritto al programma fedeltà verranno assegnati dei punti fedeltà per ogni biglietto acquistato. Il numero di punti fedeltà per biglietto varia da biglietto a biglietto.
* **Annullamento programma fedeltà:** Se un cliente iscritto al programma fedeltà non acquista biglietti aerei per due anni, verrà notificato dal sistema che un altro anno senza acquisti comporterebbe l’annullamento del programma fedeltà a lui riservato. Il sistema dopo un anno dall’invio della notifica, se il cliente non ha acquistato alcun biglietto, provvederà in automatico all’annullamento del programma fedeltà.

## 2.2 Requisiti di sistema

Per un corretto funzionamento del nostro applicativo, la macchina che lo eseguirà avrà bisogno di:

* Apache Tomcat v9.0
* Eclipse IDE for Enterprise Java Developers – versione dicembre 2019 (nel caso l’utilizzatore dell’applicativo voglia modificare il codice)
* Java Runtime Environment v8.241
* Un qualsiasi browser

# Diagramma delle classi a livello di dominio



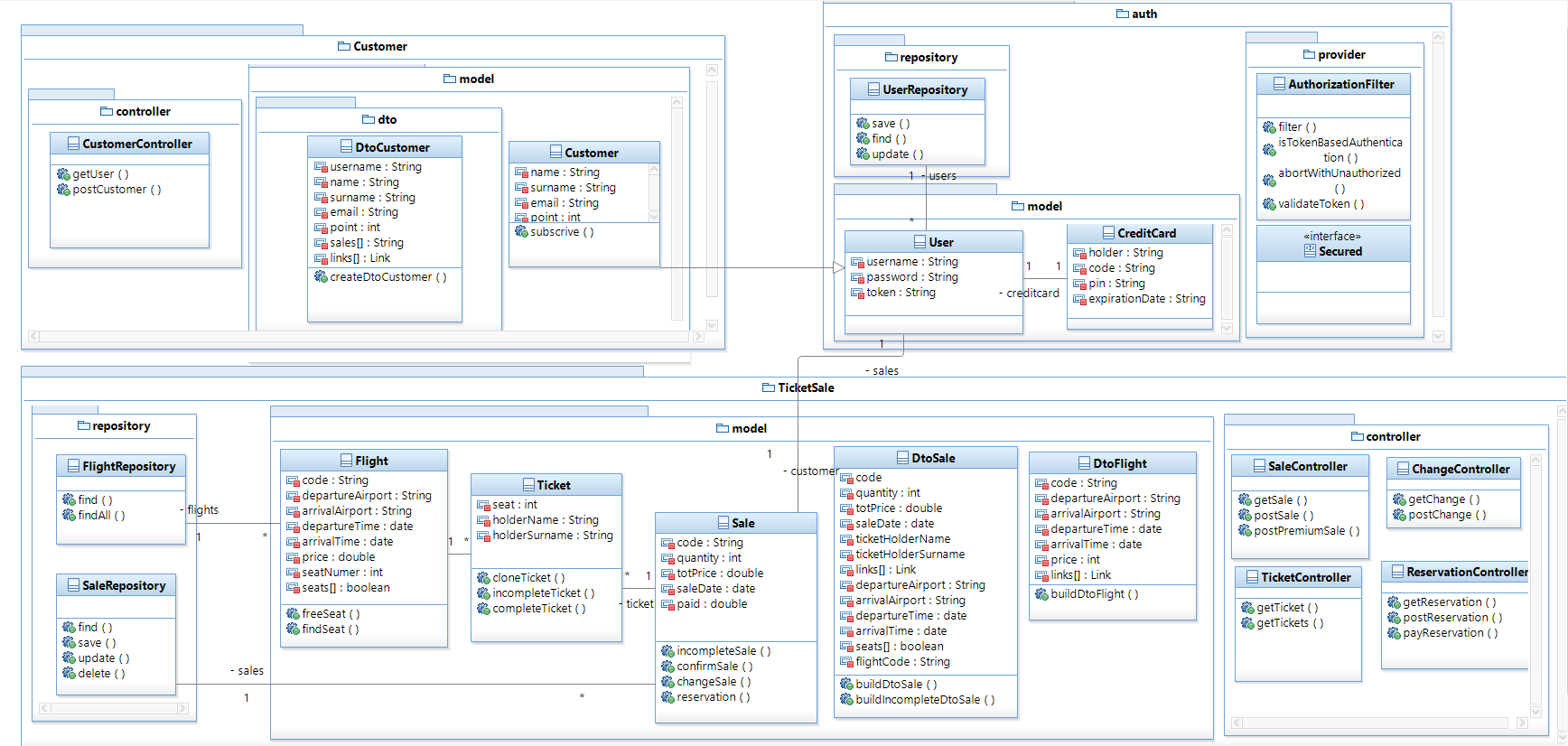
Abbiamo deciso di strutturare il programma creando tre package: **Auth, Customer e TicketSale.**

Il package **Auth** gestisce gli utenti non loggati (User) e dà ad essi la possibilità di loggarsi o registrarsi. Un oggetto di tipo **User** ha tra gli attributi username e password, quindi se un User cerca di **loggarsi** con un username e una password contenuti all’interno di UserRepository, diventa **Customer**. Al contrario se un User cerca di **registrarsi** con un username e una password già contenuti all’interno di UserRepository, la registrazione non andrà a buon fine. Inoltre il package Auth ha al suo interno anche la classe **CreditCard**, fondamentale per poter associare ad ogni utente una carta di credito, che sarà necessaria nella fase di acquisto.

Il package **Customer** contiene gestisce gli utenti. La classe **Customer** contiene tutte le informazioni utili dell’utente. La classe Customer consente all’utente loggato di **registrarsi al programma fedeltà,** inoltre la classe Customer è in relazione con la classe **User** a sua volta in relazione con la classe **Sale** e questo consente ad un utente loggato di effettuare **prenotazioni** e/o **acquisti** di biglietti aerei.

Il package **TicketSale** gestisce tutta la parte di gestione dei biglietti aerei. E’ formata da tre classi tutte associate fra di loro che sono **Flight, Ticket e Sale.** La classe **Flight** contiene tutte le informazioni di un volo, e consente di **crearne di nuovi**. Questa classe è collegata con **Ticket** per **creare dei biglietti** per quel volo, ed è collegata con **Sale** per **gestire le vendite** dei biglietti di un determinato volo. La classe **Sale** contiene tutte le informazioni di una vendita, è collegata alle classi Ticket e Flight in quanto la vendita fa riferimento ad uno o più biglietti di un determinato volo. Infine la classe **Ticket** contiene tutte le informazioni di un biglietto aereo, ed è collegata a Flight e Sale, in quanto ogni biglietto è collegato ad un volo, e può essere venduto (quindi necessita della classe Sale).

# Diagramma delle classi a livello di progettazione



All’interno del diagramma di progettazione, per rendere il tutto più leggibile ed ordinato, abbiamo deciso di non inserire dei metodi che si ripresentano in **TUTTE** le classi presenti all’interno del diagramma.

Questi metodi sono:

* Il **costruttore** della classe
* Metodi **get()** e **set()**
* Metodi **hash()** e **hashCode()**
* Metodi **equals()**

Vengono invece di seguito elencate le **funzionalità dei metodi** presenti all’interno del diagramma di progettazione:

**CLASSE UserRepository:**

**UserRepository.find(String username):** Cerca un utente all’interno della repository.  
**UserRepository.save(User user):** Salva l’utente all’interno della repository.  
**UserRepository.update(String username, User user):** Aggiorna la repository per visualizzare nuovi oggetti di tipo User appena inseriti.

**CLASSE Customer:**

**Customer.subscribe(String name, String surname, String email):** Metodo per la registrazione di un Utente al programma fedeltà.

**CLASSE CustomerController:**

**CustomerController.getUser(@HeaderParam("Authorization") String token):** Tramite il token passatogli come parametro recupera il corrispondente utente nella UserRepository, produce un oggetto DtoCustomer che ritorna come formato JSON. **CustomerController.postCustomer** **(String jsonDtoCustomer):** Trasforma l'oggetto JSON passatogli come parametro in un DtoCustomer, prende i vari attributi necessari per permettere il passaggio da utente a cliente (premium), crea un nuovo oggetto di tipo Customer e le sostituisce in UserRepository con il vecchio oggetto di tipo User, produce un nuovo DtoCustomer e lo ritorna in formato JSON.

**CLASSE DtoSale:**

**DtoSale.buildDtoSale(Sale sale):** Crea un nuovo DTO di tipo Sale.  
**DtoSale.buildIncompleteDtoSale(Sale sale):** Crea un oggetto DtoSale con attributi holdersName[quantity], holdersSurname[quantity

], e saleDate impostati a stringhe vuote.

**CLASSE Flight:**

**Flight.findSeat():** Ritorna lo stato del posto richiesto (in vendita, prenotato, acquistato, ecc…).  
**Flight.freeSeat(int seat):** Libera il posto richiesto.

**CLASSE Sale:**

**Sale.changeSale(int quantity, Ticket ticket, String [] names, String [] surnames):** Modifica un oggetto di tipo Sale.  
**Sale.confirmSale(User customer, String[] names, String [] surnames):** Conferma gli attributi di un oggetto di tipo Sale, e ne inizializza la vendita.  
**Sale.incompleteSale(String code, int quantity, Ticket ticket):** Crea un oggetto Sale con attributi holdersName[quantity], holdersSurname[quantity] e saleDate impostati come stringa vuota **Sale.reservation(User customer, String [] names, Strings [] surnames):** Assegna ad uno o più ticket name e surname.

**CLASSE Ticket:**

**Ticket.cloneTicket(Ticket ticket):** Clona un oggetto di tipo Ticket.  
**Ticket.incompleteTicket(Flight flight):** Crea un oggetto di tipo Ticket con attributi holderName e holderSurname a null e seat a 0. **Ticket.completeTicket(int seat, String holderName, String holderSurname):** Assegna i parametri seat, holderName e holderSurname.

**CLASSE SaleRepository:**

**SaleRepository.find(String code):** Trova un oggetto di tipo Sale contenuto nella repository.  
**SaleRepository.save(Sale sale):** Salva tutti gli oggetti di tipo Sale all’interno della repository.  
**SaleRepository.update(String code, Sale sale):** Aggiorna la repository per visualizzare nuovi oggetti di tipo Sale appena inseriti.  
**SaleRepository.delete(String code):** Cancella un determinato oggetto di tipo Sale.

**CLASSE FlightRepository:**

**FlightRepository.find(String code):** Trova un oggetto di tipo Flight contenuto nella repository.  
**FlightRepository.findall():** Trova tutti gli oggetti di tipo Flight contenuti nella repository.

**CLASSE SaleController:**

**SaleController.getSale(@PathParam("code") String code, @QueryParam("quantity") int quantity):** Genera un ticket (incompleto) basandosi sul codice del volo presente nell'url, il ticket viene utilizzato per inizzializzare un oggetto di tipo sale (incompleto); questo viene prodotto aggiungendo all'oggetto sale un numero di ticket (incompleti) equivalente al parametro query quantity (se presente nell'url) o aggiungendo un solo ticket se il parametro quantity non è presente, calcola il prezzo totale di sale come la somma dei prezzi di ogni ticket (attraverso il metodo Sale.incompleteSale(String code, int quantity, Ticket ticket)).  
**SaleController.postSale(@HeaderParam("Authorization") String token, String jsonDtoSale):** Recupera l'oggetto sale da SaleRepository tramite l'attributo code dell'oggetto DtoSale che gli viene passato come parametro in formato JSON, recupera le informazioni dell'utente da UserRepository tramite il parametro token prelevato dall'header della richiesta; aggiunge queste informazioni all'oggetto sale precedentemente recuperato insieme a tutte le altre informazioni per il completamento del biglietto, ricrea un DtoSale basandosi sull'oggetto sale precedentemente modificato e genera una risposta in formato JSON.  
**SaleController.postPremiumSale(@HeaderParam("Authorization") String token, String jsonDtoSale):** Ha lo stesso procedimento di postSale(), ma per utilizzarla bisogna essere utenti premium (Customer) e avere abbastanza punti fedeltà per completare l'acquisto.

**CLASSE TicketController:**

**TicketController.getTickets():** Recupera la lista di voli da FlightRepository e genera una nuova lista di DtoFlight utilizzando gli oggetti Flight presenti nella repository, per ogni oggetto DtoFlight generato crea un link alle nuove risorse accessibili.  
**TicketController.getTicket(@PathParam("code") String code):** Recupera un singolo volo da FlightRepository basandosi sul codice presente nell'url, genera un DtoFlight del volo e i link alle nuove risorse accessibili.

**CLASSE ChangeController:**

**ChangeController.postChange(@PathParam("code") String code, String jsonDtoSale):** Genera un ticket (incompleto) basandosi sul codice del volo presente nell'url, recupera l'oggetto sale da SaleRepository tramite l'attributo code dell'oggetto DtoSale che gli viene passato come parametro in formato JSON, modifica il sale utilizzando i dati presenti in DtoSale (holdersName[], holdersSurname) e il ticket generato in precedenza, salva il nuovo acquisto e ritorna un DtoSale in formato JSON.  
**ChangeController.getChange(@PathParam("salecode") String saleCode):** Ricerca un sale precedentemente acquistato nella SaleRepository utilizzando il codice presente nell'url, produce un DtoSale che manda come risposta in formato JSON.

**CLASSE ReservationController:**

**ReservationController.getReservation(@PathParam("code") String code, @QueryParam("quantity") int quantity):** Genera un ticket (incompleto) basandosi sul codice del volo presente nell'url, il ticket viene utilizzato per inizzializzare un oggetto di tipo sale (incompleto); questo viene prodotto aggiungendo all'oggetto sale un numero di ticket (incompleti) equivalente al parametro query quantity (se presente nell'url) o aggiungendo un solo ticket se il parametro quantity non è presente, calcola il prezzo totale di sale come la somma dei prezzi di ogni ticket (attraverso il metodo Sale.incompleteSale(String code, int quantity, Ticket ticket)).  
**ReservationController.postReservation(@HeaderParam("Authorization") String token, String jsonDtoSale):** Recupera l'oggetto sale da SaleRepository tramite l'attributo code dell'oggetto DtoSale che gli viene passato come parametro in formato JSON, recupera le informazioni dell'utente da UserRepository tramite il parametro token prelevato dall'header della richiesta; aggiunge queste informazioni all'oggetto sale precedentemente recuperato insieme a tutte le altre informazioni per il completamento del biglietto, (ma mantiene l'attributo paid = false) ricrea un DtoSale basandosi sull'oggetto sale precedentemente modificato e genera una risposta in formato JSON.  
**ReservationController.payReservation(@@PathParam("reservationcode") String code):** Recupera la prenotazione non ancora pagata, completa il pagamento della prenotazione e ritorna un oggetto DtoSale in formato JSON.

**CLASSE AuthenticationEndpoint:**

**AuthenticationEndpoint.authenticateUser(@FormParam("username") String username, @FormParam("password") String password):** Utilizza il parametro username per recuperare l'oggetto User nella UserRepository, controlla che la password del parametro sia uguale alla password dell'oggetto recuperato e se lo e genera un nuovo token che ritorna come risposta in formato JSON altrimenti risponde alla richiesta con lo stato Http 403.

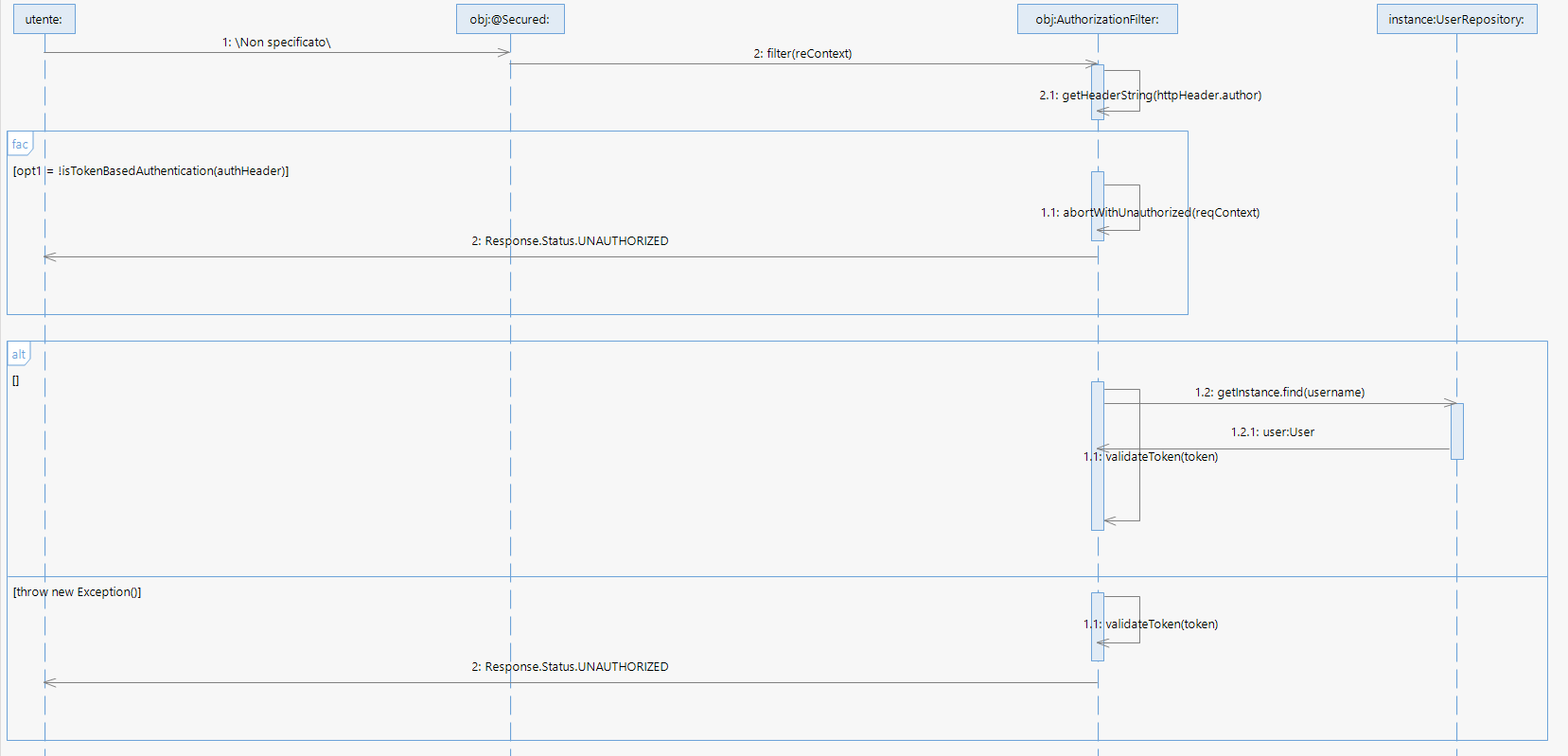
**CLASSE RegistrationEndpoint:**

**RegistrationEndpoint.registerUser(@FormParam("username") String username, @FormParam("password") String password) :** Utilizza l'username e la password passati come parametri per generare un nuovo oggetto User che salva in UserRepository, genera un nuovo token che ritorna come risposta in formato JSON altrimenti risponde alla richiesta con lo stato Http 403 nel caso l'username sia già esistente in UserRepository.

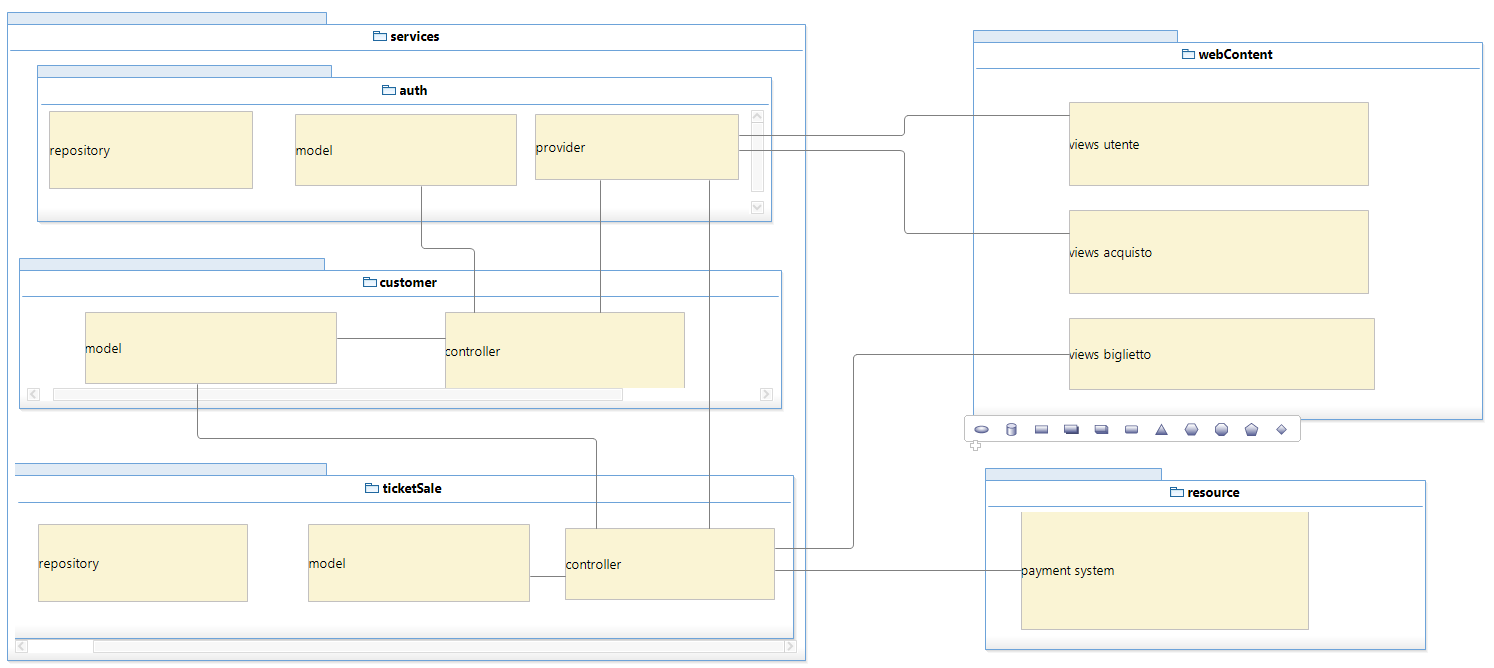
**CLASSE AuthorizationFilter:**

**AuthorizationFilter.filter(ContainerRequestContext requestContext) :** Recupera il token presente nell'header, controlla che il token corrisponda al token presente nell'oggetto User dell'utente (che viene recuperato dalla UserRepository tramite la prima parte del token) che ha fatto la richiesta, se corrisponde il filtro garantisce la richiesta alla risorsa richiesta dall'utente altrimenti risponde con uno stato Http 401.

# Diagramma di sequenza

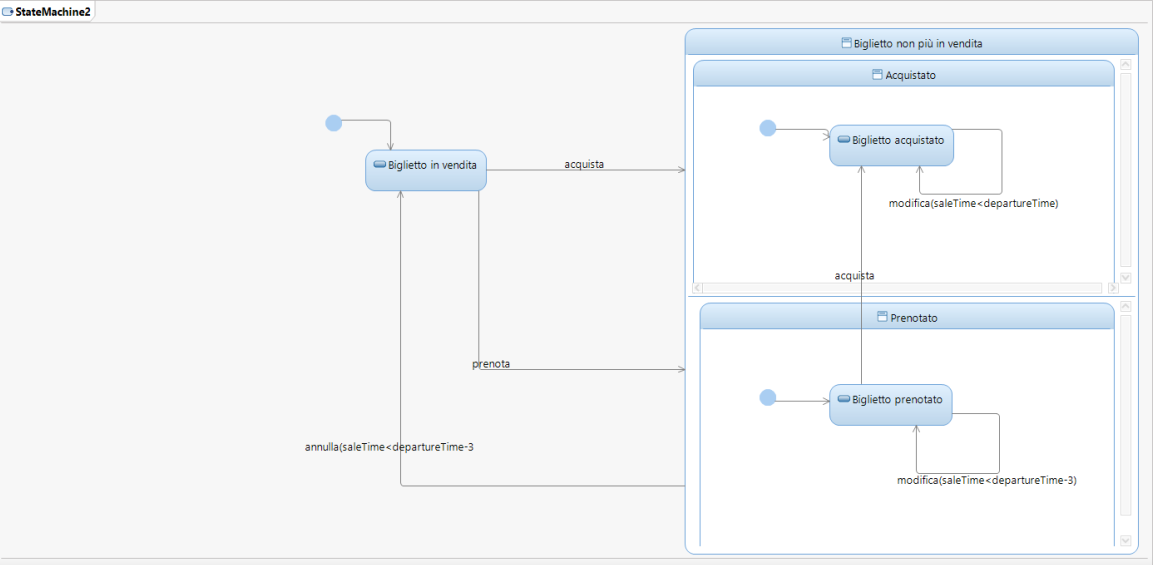


# Diagramma di architettura software



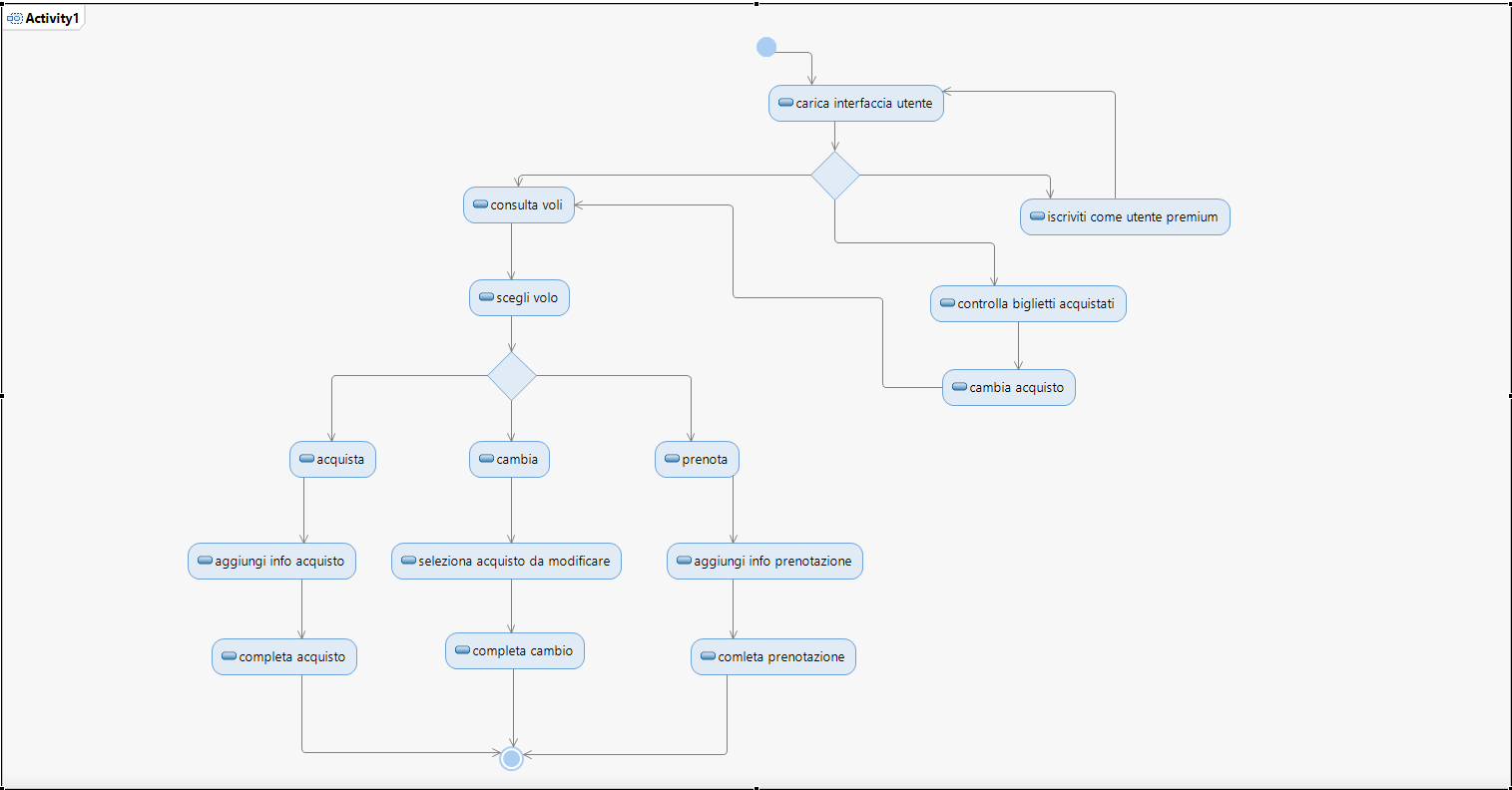
In questo diagramma è possibile notare quanto sia stato incidente l’utilizzo di **MVC** come pattern. Possiamo infatti notare come ogni package abbia al suo interno una classe **controller** e una **model**. Il package **services**, che comprende tutta la parte di codice **Javascript** si unirà poi al package **webContent** che invece gestisce la parte **html** dell’applicativo.

# Diagramma degli stati



Il seguente diagramma degli stati rappresenta l’elemento ticket. Infatti un biglietto può passare da tre stati diversi: **biglietto in vendita**, **biglietto acquistato** e **biglietto prenotato** (questi ultimi due stati fanno parte dello stato più generico **“biglietto non più in vendita”**).

# Diagramma di attività



Qui sopra abbiamo un diagramma delle attività piuttosto generico sull’uso dell’applicativo da parte di un cliente/utente.

# Pattern utilizzati

## MVC (Model-View-Controller)

Un pattern che è stato largamente utilizzato nel nostro progetto è stato indubbiamente MVC.

Abbiamo scelto di utilizzare MVC perché ci risultava una buona soluzione viste le richieste del progetto. Avevamo bisogno di un’architettura nella quale fosse facile ed immediato modificare degli oggetti di vario tipo (Flight, Ticket, Customer, ecc…) e l’MVC si presta bene a questo compito, oltre a riuscire a visualizzare altrettanto facilmente tutti gli oggetti di vario tipo.

## DTO (Data Transfer Object)

Abbiamo inoltre utilizzato anche il pattern architetturale DTO per la comunicazione e il passaggio di dati tra server e client.

DTO è un oggetto che sposta dei dati attraverso processi. Questo pattern all’interno del nostro applicativo viene usato per passare degli oggetti da lato server a lato client, come ad esempio durante la consultazione dei voli disponibili. Il client chiederà di visualizzare una lista di voli con certe caratteristiche (aeroporto di partenza, data di partenza, ecc…), a questo punto dalla repository Flight otterremo una lista di oggetti “validi” che verranno passati al client come DTO.

## Repository

E’ stato usato inoltre il pattern repository, in alternativa ad un database, per immagazzinare tutti gli oggetti di vario tipo (Flight, Ticket, Customer, ecc…).