Sap-Assigment01 Report

Shipping on the Air

Bedeschi Federica
0001175655 federica.bedeschi4@studio.unibo.it

Pracucci Filippo
0001183555 filippo.pracucci@studio.unibo.it

Anno accademico 2025-2026

Indice

Analisi

1.1 Descrizione del problema

Il sistema di consegna di pacchi tramite droni deve permettere agli utenti di:

- inviare pacchi (di un certo peso) da un posto a un altro, consentendo la spedizione in un determinato momento (anche immediatamente) entro un periodo massimo;
- tracciare la consegna, in modo da conoscerne lo stato e il tempo rimanente.

Per usufrire del sistema è necessario registrarsi ed effettuare il login prima di ogni sessione di utilizzo.

1.2 Processo di analisi

Per prima cosa abbiamo analizzato i requisiti del sistema specificandone i **casi d'uso** e li abbiamo raccolti e rappresentati tramite il diagramma UML in Figura 1.1.

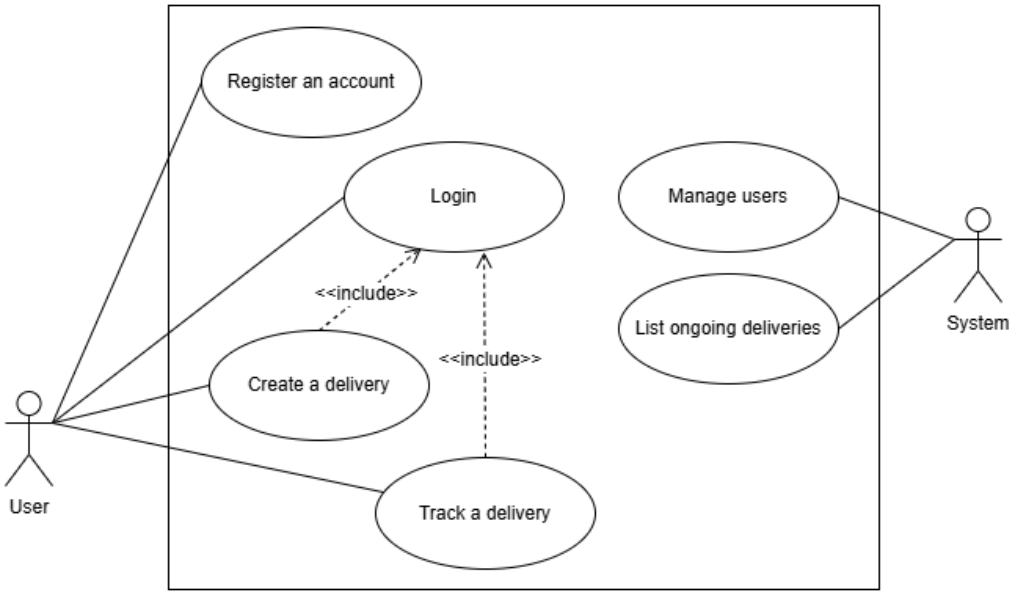

Figura 1.1: Diagramma dei casi d'uso

Partendo da questi abbiamo sviluppato le **User Stories**:

```
As a person,
I want to create an account
so that I can login and use the system for delivering
   packages.

As a person,
I want to login
so that I can access the system for delivering packages.

As a user,
I want to send a package
so that I can deliver objects to other places.

As a user,
I want to track the delivering process
so that I can know the process' state and the time left to
   complete the delivery.
```

Da ognuna abbiamo derivato i relativi **Acceptance Criteria** sotto forma di scenari. Un esempio è:

```
Scenario: Successful registration
    Given I am on the registration page
    And I have not an account
```

```
When I create an account with a username "marco" and a
   valid password "Secret#123"
Then I should see a confirmation that my account has been
    created and receive my identifier
```

In stile **Behavior-Driven Development (BDD)**, abbiamo raccolto tutti gli scenari e li abbiamo utilizzati per effettuare **Acceptance Tests** utilizzando il tool **Cucumber**.

1.2.1 DDD

Come approccio metodologico abbiamo utilizzato il **DDD**, e come prima cosa abbiamo definito l'**Ubiquitous Language**, ad esempio:

- *Delivery state*: stato corrente della spedizione, come `READY_TO_SHIP`, `SHIPPING`, `DELIVERED`;
- *Delivery status*: composizione del DeliveryState e del tempo rimanente alla consegna;
- *Place*: indirizzo composto da via e numero civico, utilizzato come punto di partenza e di arrivo;
- *Expected shipping moment*: la data e l'orario specificati dall'utente per l'invio del pacco.

Successivamente, abbiamo modellato il dominio dell'applicazione identificandone i concetti e collegandoli ai relativi **Building Blocks** (entities, value objects, aggregates, etc.). I collegamenti sono stati effettuati all'interno del codice tramite l'estensione delle interfacce rappresentanti i concetti con quelle dei building blocks.
Durante la modellazione abbiamo identificato due **Bounded Contexts** principali:

- *Users context*: contesto che racchiude i concetti legati agli utenti del sistema;
- *Deliveries context*: contesto che racchiude i concetti legati alle spedizioni, nello specifico la loro creazione e il loro tracciamento; comprende anche i droni utilizzati nelle spedizioni.

Design e architettura

Abbiamo utilizzato un'architettura a microservizi, dove per ognuno dei quali abbiamo utilizzato un'architettura esagonale. Abbiamo realizzato tre microservizi: account service, lobby service e delivery service. Ognuno di questi espone un'interfaccia di tipo **REST API**, documentate tramite *OpenApi* come mostrato in Figura 2.1, per interagire con il sistema.

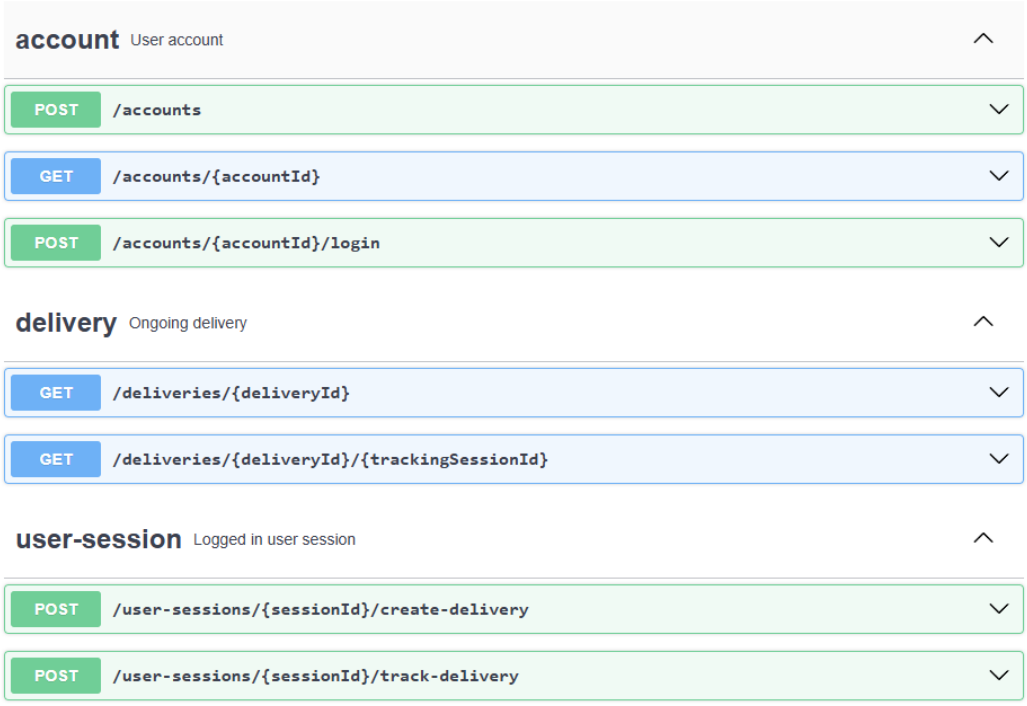

Figura 2.1: Visualizzazione tramite Swagger della specifica OpenApi

2.1 Account service

L'account service si occupa della gestione degli account, permettendone la registrazione e garantendo la loro persistenza tramite repository su file. L'architettura del microservizio segue quella mostrata in Figura 2.2, che evidenzia l'utilizzo di `AccountService` come *inbound port* e `AccountRepository` come *outbound port*, la quale viene implementata tramite l'*adapter* `FileBasedAccountRepository`.

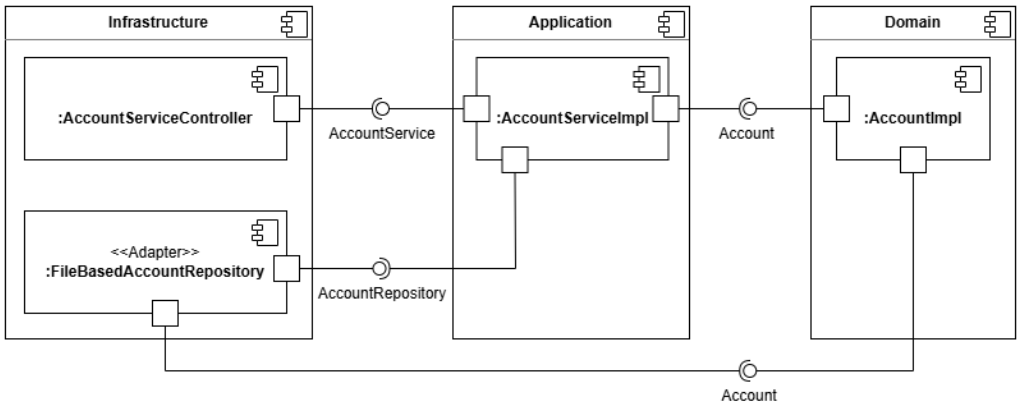

Figura 2.2: C&C diagram

2.2 Lobby service

Il lobby service si occupa della gestione delle sessioni degli utenti e delle azioni di login (comunicando con l'account service), di creazione e di tracciamento di delivery (comunicando con il delivery service). La comunicazione con gli altri microservizi avviene tramite *proxy*, effettuando delle richieste HTTP basandosi sulle REST API messe a disposizione. Questo è stato gestito in maniera sincrona, pur consapevoli del fatto che esista la possibilità che si possa bloccare l'*event-loop* di *Vert.x*, rimanendo in attesa di una risposta. Le sessioni degli utenti non vengono mantenute in maniera persistente in quanto temporanee. L'architettura del microservizio segue quella mostrata in Figura 2.3, che evidenzia l'utilizzo di `LobbyService` come *inbound port*, mentre `AccountService` e `DeliveryService` come *outbound port*, le quali vengono implementate tramite gli *adapter* `AccountServiceProxy` e `DeliveryServiceProxy`.

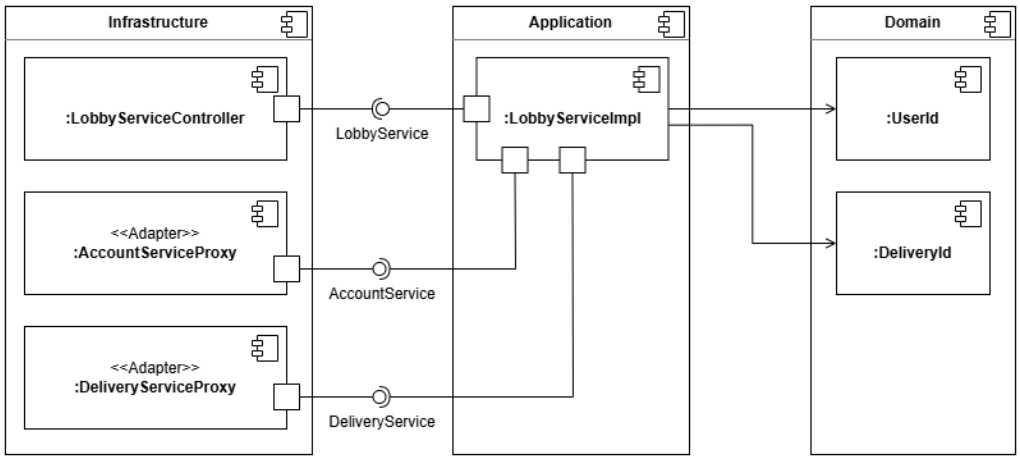

Figura 2.3: C&C diagram

2.3 Delivery service

Il delivery service ha il compito di gestire le delivery, permettendone la creazione, garantendo la loro persistenza tramite repository su file, ed il tracciamento. Internamente utilizza dei droni per effettuare le spedizioni, i quali agiscono come se fossero entità autonome (ogni drone è un *Virtual Thread*) ed inviano eventi per notificare aggiornamenti nel processo. Inoltre sfrutta delle `TrackingSessions` per modellare il fatto che un utente stia tracciando una delivery; analogamente alle sessioni degli utenti, non si ha persistenza neanche delle sessioni di tracciamento. Il servizio modella il proprio comportamento tramite un'architettura ad eventi e sfruttando il pattern **Observer**. I **domain events** (`Shipped`, `TimeElapsed` e `Delivered`) vengono generati dai droni, i quali notificano le `Delivery` che a loro volta inoltrano l'evento al `DeliveryService` ed all'utente nel caso in cui stia tracciando la delivery. In quest'ultima situazione a livello di infrastruttura abbiamo utilizzato *Vert.x* e le *web sockets* per creare un canale di comunicazione con l'utente. L'architettura del microservizio segue quella mostrata in Figura 2.4, che evidenzia l'utilizzo di `DeliveryService` come *inbound port*, mentre `DeliveryRepository` e `TrackingSessionEventObserver` come *outbound port*, le quali vengono implementate tramite gli *adapter* `FileBasedDeliveryRepository` e `VertxTrackingSessionEventObserver`.

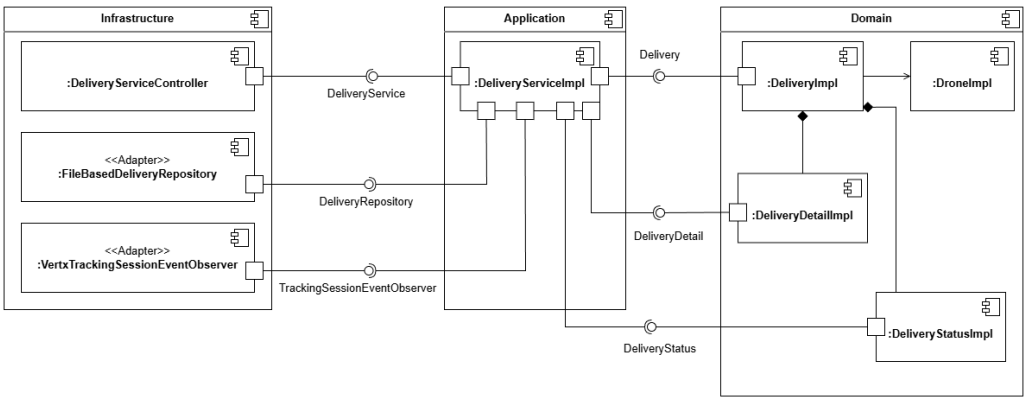

Figura 2.4: C&C diagram

Testing

Abbiamo realizzato alcuni **Unit tests** per testare i componenti principali dei tre microservizi. Inoltre abbiamo sfruttato dei test architetturali per verificare che ogni microservizio rispetti lo stile architetturale esagonale. Infine abbiamo implementato alcuni **Acceptance Tests**, sfruttando il tool *Cucumber*. Nello specifico abbiamo verificato le *user stories* riguardanti la registrazione, il login, la creazione di una delivery e il tracciamento di delivery; per ognuna abbiamo creato sia scenari di successo che di fallimento, dove necessario.

Conclusioni

L'utilizzo del DDD ha portato diversi vantaggi, specialmente nella fase di analisi e raccolta di requisiti, facendoci concentrare inizialmente sulla *business logic* astraendo dagli aspetti tecnologici ed implementativi. La creazione di *user stories* e della specifica *OpenApi* ha favorito l'individuazione delle risorse e delle azioni che il sistema doveva modellare, inclusi i comportamenti attesi, oltre che fornirci una base per la realizzazione di test sulle funzionalità del sistema. L'architettura a microservizi ci ha consentito di concentrarci su un singolo *bounded context* alla volta, facilitandone il design e la conseguente implementazione. Inoltre l'architettura esagonale adottata per i singoli microservizi ci ha permesso di separare gli aspetti legati al dominio da quelli tecnologici ed implementativi, specialmente evitando dipendenze del dominio da questi ultimi aspetti.