

Università di Bologna
Dipartimento di Informatica — Scienza e Ingegneria (DISI)
Corso di Laurea Magistrale in Informatica

ACME Mobility
**Architettura a Microservizi per un Servizio
di Noleggio di Veicoli Elettrici**

Corso: Architetture Software a Microservizi
Anno Accademico: 2025/2026

Alessandro Campedelli	0001242904
Nicolas Cola	0001235951
Francesco Maria Fuligni	0001246029

Indice

1	Introduzione	3
1.1	Obiettivo del progetto	3
1.2	Breve descrizione del dominio ACMEMobility	3
1.3	Tecnologie adottate	4
1.4	Risorse e artefatti del progetto	4
2	Coreografia e proiezione dei ruoli	6
2.1	Partecipanti della coreografia	6
2.2	Coreografia globale	6
2.2.1	Fase iniziale	6
2.2.2	Scenario A: Noleggio Immediato	7
2.2.3	Monitoraggio durante la corsa	7
2.2.4	Scenario B: Prenotazione Breve	8
2.2.5	Fase di chiusura comune	8
2.3	Discussione della connectedness e raffinamenti	9
2.3.1	Definizione	9
2.3.2	Analisi sulla coreografia di ACME Mobility	10
2.4	Proiezioni locali dei ruoli	11
3	Modellazione dei processi BPMN	12
3.1	Diagramma di collaborazione documentativo	12
3.2	Processo di noleggio eseguibile	13
3.2.1	Lettura sequenziale del processo	13
3.2.2	Prenotazione: eventi alternativi e compensazione	14
3.2.3	Esiti terminali: manutenzione e ricarica	14
3.3	Processi complementari al noleggio	15
3.3.1	Processo di manutenzione (<code>maintenance_process.bpmn</code>)	15
3.3.2	Processo di aggiornamento stato (<code>update_status.bpmn</code>)	16
3.4	Coerenza fra BPMN documentativo e BPMN eseguibili	16
3.4.1	Divergenza nel monitoraggio	17
4	Progettazione della SOA	18
4.1	Diagramma UML/TinySOA	18
4.2	Comunicazioni tra servizi	20
4.2.1	REST (HTTP/JSON)	20
4.2.2	SOAP con Jolie	20
4.2.3	gRPC	20
4.2.4	BPMS — Camunda/Zeebe	21

5	Realizzazione e deployment	23
5.1	Implementazione con Camunda e worker	23
5.1.1	Architettura modulare del Rental Service	23
5.1.2	Organizzazione dei Job Worker e orchestrazione Zeebe	23
5.1.3	Navigazione guidata dal BPMN	24
5.2	Banca Jolie	25
5.2.1	Architettura a doppio servizio (SOAP e REST)	25
5.2.2	Interfaccia SOAP e modello transazionale	25
5.2.3	Gestione dello stato contabile e correlazione	26
5.3	Fleet Management come microservizi	26
5.3.1	Decomposizione a microservizi e protocolli di comunicazione	26
5.3.2	Simulazione del percorso con GraphHopper	27
5.3.3	Aggregazione telemetrica e streaming SSE	27
5.4	Stations Service	28
5.4.1	Catalogo e operazioni sui veicoli	28
5.4.2	Ricarica asincrona e callback	29
5.5	Maintenance Service	29
5.5.1	Coda ed elaborazione programmata	29
5.5.2	Ricostruzione della coda	29
5.6	Containerizzazione con Docker Compose	29
5.6.1	Struttura del Compose	29
5.6.2	Health check e dipendenze	30
5.6.3	Volumi	30
5.7	Istruzioni minime di build e avvio	30
5.7.1	Requisiti	30
5.7.2	Primo avvio	30
5.7.3	Account demo	31
5.7.4	Avvii successivi e pulizia	31
5.8	Interfaccia utente	31
6	Conclusioni	33
6.1	Risultati raggiunti	33
6.2	Limiti e semplificazioni adottate	33

1 Introduzione

1.1 Obiettivo del progetto

Questo documento descrive il lavoro svolto nell'ambito del progetto del corso di *Architetture Software a Microservizi* (A.A. 2025/2026). L'obiettivo è la progettazione e l'implementazione di **ACME Mobility**, una piattaforma di noleggio a breve termine di veicoli elettrici distribuiti in stazioni cittadine.

Il progetto copre l'intero ciclo richiesto dalla specifica:

1. Modellazione della **coreografia globale** tra i partecipanti, analisi della *connectedness* e proiezione locale per ogni ruolo.
2. Modellazione **BPMN** documentativa (diagramma di collaborazione) ed eseguibile (processi Camunda).
3. **Progettazione SOA** con diagramma UML/profilo TinySOA.
4. **Implementazione e deployment** con Camunda 8, Jolie (SOAP), API REST, gRPC e Docker Compose.

1.2 Breve descrizione del dominio ACMEMobility

ACMEMobility propone ai propri clienti un servizio di noleggio a breve termine di veicoli elettrici (auto, scooter, monopattini) dislocati in stazioni cittadine. I clienti interagiscono tramite un'applicazione web.

Il sistema supporta due modalità operative:

Noleggio Immediato

Il cliente individua un veicolo disponibile sulla mappa e ne simula la scansione del QR code per avviare il noleggio.

Prenotazione Breve

Il cliente prenota un veicolo con un anticipo massimo di 30 minuti. Il veicolo viene bloccato digitalmente fino al ritiro.

Il pagamento è strutturato in due fasi: una *pre-autorizzazione* (blocco cauzionale di 10€) all'avvio e un *pagamento finale* a fine corsa, calcolato su tempo e tariffa per categoria, con penale del 10% se la batteria è sotto il 15%.

Al termine del noleggio, dopo aver ricevuto il riepilogo della corsa, l'utente può inviare un *report* indicando gli eventuali danni riscontrati sul veicolo. Una segnalazione comporta l'inserimento del mezzo nel flusso di manutenzione; in assenza di danni, il sistema valuta invece se sia necessaria la ricarica prima di renderlo nuovamente disponibile.

L'implementazione include 7 stazioni reali a Bologna (con coordinate geografiche reali) e una flotta iniziale di 99 veicoli (33 per categoria).

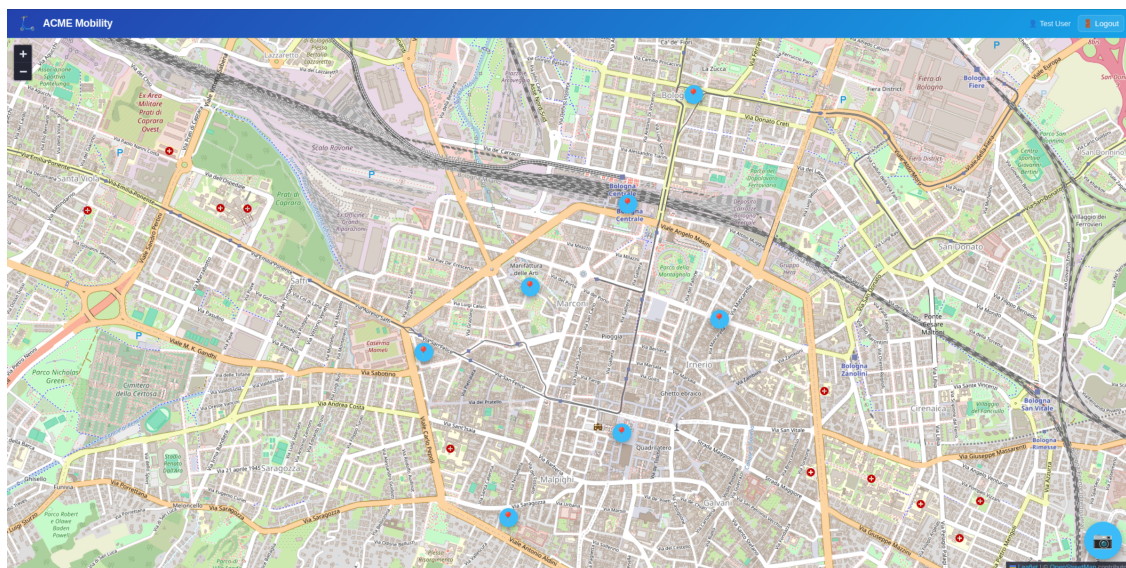


Figura 1: Mappa con le 7 stazioni di Bologna, veicoli disponibili e indicatori di stato.

1.3 Tecnologie adottate

La tabella 1 riassume le principali tecnologie del progetto.

Il dialogo tra BPMS e banca avviene via **SOAP**. Fleet Management espone un gateway **REST** e comunica internamente tramite **gRPC**. Tutti i servizi sono containerizzati con Docker Compose.

1.4 Risorse e artefatti del progetto

La presente relazione mantiene il carattere di diario operativo del progetto: documenta il percorso seguito, le principali scelte progettuali e il rapporto fra i modelli prodotti e la loro implementazione. Gli artefatti tecnici aggiornati, le specifiche, i diagrammi e la documentazione di supporto sono consultabili nella GitHub Pages del progetto:

GitHub Pages del progetto ACME Mobility

Questa separazione evita di duplicare nella relazione contenuti soggetti a evoluzione e mantiene la documentazione tecnica consultabile in un unico punto aggiornato.

Tabella 1: Tecnologie adottate nel progetto

Componente	Tecnologia	Ruolo
Frontend	Angular 21, Leaflet/OSM	UI, mappa, prenotazioni, monitoraggio corsa
Rental Service	Java 21, Spring Boot 3	API pubbliche, stato dominio, worker Camunda, SSE
Orchestratore	Camunda 8.9.1 / Zeebe	Esecuzione processi BPMN, correlazione eventi
Bank Service	Jolie, SOAP	Blocco/sblocco cauzione, addebito
Account Registration	Jolie, REST	Creazione account bancari
Stations Service	Java 21, Spring Boot	Catalogo stazioni, sblocco/blocco/ricarica veicoli
Maintenance Service	Java 21, Spring Boot	Coda manutenzione e simulazione
Fleet Management Gateway	Java 21, Spring Boot, gRPC	Monitoraggio posizione e batteria (REST + gRPC)
Tracking Service	Java 21, Spring Boot, gRPC	Stato posizione veicoli in noleggio
Battery Service	Java 21, Spring Boot, gRPC	Stato batteria veicoli in noleggio
DB Rental	PostgreSQL 18	Utenti, sessioni BPMN, flotta, noleggi
DB Bank	PostgreSQL 18	Conti, saldi, importi riservati, token
Messaging	Redis Pub/Sub + SSE	Notifiche real-time al frontend
Containerizzazione	Docker Compose	Build, deploy, networking, volumi
Route planning	GraphHopper	Calcolo percorsi reali tra stazioni

2 Coreografia e proiezione dei ruoli

2.1 Partecipanti della coreografia

La coreografia globale coinvolge otto partecipanti:

RS — Rental Service

Orchestratore principale del sistema ACME Mobility. Implementato come BPMS (Camunda). Riceve le richieste dell'utente, coordina banca, stazioni e Fleet Management, e mantiene lo stato di noleggi e prenotazioni.

User — Cliente

L'utente finale che interagisce tramite l'applicazione web. Avvia noleggi e prenotazioni, monitora la corsa, termina il noleggio e segnala eventuali danni.

B — Bank

Servizio bancario terzo, implementato in Jolie con interfaccia SOAP. Gestisce blocchi cauzionali, addebiti definitivi e sblocchi.

S — Station

Gestisce fisicamente i veicoli nelle stazioni: sblocco, blocco e ricarica. Espone API REST.

FM — Fleet Management

Coordinatore del monitoraggio dei veicoli in corsa. Espone un gateway REST e comunica internamente con Tracking e Battery via gRPC.

TS — Tracking Service

Microservizio responsabile del tracciamento della posizione geografica dei veicoli. Comunicazione via gRPC.

BMS — Battery Monitoring Service

Microservizio responsabile del monitoraggio del livello di batteria. Comunicazione via gRPC.

MS — Maintenance Service

Gestisce la coda di manutenzione. Riceve notifiche di veicoli danneggiati e coordina le operazioni di ripristino.

2.2 Coreografia globale

La coreografia globale descrive formalmente tutte le interazioni tra i partecipanti. È strutturata in due macro-scenari (**Scenario A**: Noleggio Immediato; **Scenario B**: Prenotazione Breve) composti con l'operatore di scelta esterna (+).

2.2.1 Fase iniziale

```
1 map_opening: User -> RS;
```

```
2 return_vehicles: RS -> User;
```

Listing 1: Fase iniziale

2.2.2 Scenario A: Noleggio Immediato

Il cliente scansiona il QR code. Il sistema procede con la pre-autorizzazione bancaria (blocco 10€), avvio del monitoraggio FM (tracking e batteria in parallelo), sblocco fisico del veicolo, loop di monitoraggio durante la corsa, e infine la chiusura: blocco veicolo, stop monitoraggio, pagamento finale, sblocco cauzione, riepilogo e gestione danni/ricarica.

```
1 scanning: User -> RS;
2
3 block_money: RS -> B;
4 (
5   // RAMO A1: Happy Path
6   (
7     send_token: B -> RS;
8
9     start_monitoring: RS -> FM;
10    (
11      (start_tracking: FM -> TS; ack_start_tracking: TS -> FM)
12      |
13      (start_battery_monitoring: FM -> BMS;
14        ack_start_battery_monitoring: BMS -> FM)
15    );
16    ack_monitoring: FM -> RS;
17
18    unlock_vehicle: RS -> S;
19    vehicle_unlocked: S -> RS;
20    ack_rental_started: RS -> User;
21    ...
22  )
23  +
24  // RAMO A2: Fondi Insufficienti
25  (
26    send_error: B -> RS;
27    send_error_message: RS -> User
28  )
29 )
```

Listing 2: Scenario A — pre-autorizzazione e avvio monitoraggio

2.2.3 Monitoraggio durante la corsa

Dopo la conferma dell'avvio del noleggio, la coreografia entra in un ciclo di monitoraggio che rimane attivo fino alla ricezione di `end_ride`. A ogni iterazione, l'utente invia una

richiesta di aggiornamento a RS; quest'ultimo attiva una nuova lettura presso FM, che interroga in parallelo TS e BMS per ottenere rispettivamente posizione e livello della batteria. I dati raccolti vengono quindi restituiti a RS e inoltrati all'utente.

```

1 (
2   request_update: User -> RS;
3   trigger_read: RS -> FM;
4   (
5     (request_position: FM -> TS;
6       send_position_updated: TS -> FM)
7     |
8     (request_battery_status: FM -> BMS;
9       send_battery_status_updated: BMS -> FM)
10  );
11  telemetry_update: FM -> RS;
12  vehicle_status_updated: RS -> User
13 )*;

```

Listing 3: Loop coreografico di monitoraggio

Nel BPMN eseguibile questa fase è associata al sottoprocesso di monitoraggio e al job type `monitoringLogic`. Le differenze tra l'astrazione coreografica e la sua realizzazione temporizzata saranno considerate nella discussione sulla coerenza tra i modelli (Sezione 3.4).

2.2.4 Scenario B: Prenotazione Breve

Dopo la conferma della prenotazione, il sistema si trova in uno dei tre rami mutuamente esclusivi:

B1.1 — Annullamento esplicito: Cancellazione libera (> 5 min) con sblocco cauzione, oppure penale (< 5 min) con addebito.

B1.2 — Timeout (no-show): Timer 30 min scaduto senza ritiro: la cauzione viene addebitata.

B1.3 — Ritiro del veicolo: L'utente scansiona il veicolo prenotato; il flusso prosegue identico allo Scenario A dal momento della scansione in poi.

2.2.5 Fase di chiusura comune

```

1 lock_vehicle: RS -> S; vehicle_locked: S -> RS;
2
3 stop_monitoring: RS -> FM;
4 (
5   (stop_tracking: FM -> TS; ack_stop_tracking: TS -> FM)
6   |
7   (stop_battery_monitoring: FM -> BMS;

```

```

8   ack_stop_battery_monitoring: BMS -> FM)
9   );
10  ack_stop_monitoring: FM -> RS;
11
12  request_charge_money: RS -> B; ack_charge_sent: B -> RS;
13  request_unlock_money: RS -> B; ack_money_unlocked: B -> RS;
14
15  rental_summary: RS -> User;
16  (
17    (send_report: User -> RS; vehicle_in_queue: RS -> MS; ...)
18    +
19    (no_report_written: User -> RS;
20      (recharge_request: RS -> S; ...) + (no_recharge_needed: RS ->
21        S; ...))
22  )

```

Listing 4: Chiusura del noleggio

2.3 Discussione della connectedness e raffinamenti

2.3.1 Definizione

Una coreografia è *connected* se soddisfa tre condizioni necessarie per garantire che la proiezione sui ruoli produca un sistema che si comporta conformemente alla coreografia globale¹:

Connectedness per la sequenza. Per ogni composizione sequenziale $C; C'$, ogni interazione finale di C e ogni interazione iniziale di C' devono condividere almeno un partecipante. Questo garantisce che la sincronizzazione tra i due blocchi avvenga attraverso un ruolo comune, senza richiedere coordinazione esplicita aggiuntiva.

Connectedness per la scelta. Per ogni scelta $C + C'$, il *sender* del primo messaggio deve essere lo stesso in entrambi i rami. Inoltre, ogni ruolo che compare in un solo ramo non deve rimanere bloccato in attesa di messaggi che non arriveranno mai; in pratica, ruoli non coinvolti altrove nella coreografia possono proiettare a 1 (skip) senza violare la condizione.

Causality-safety. Più comunicazioni sulla stessa operazione tra gli stessi partecipanti, abilitate contemporaneamente, non devono potersi interferire: devono essere in sequenza (abilitate una dopo l'altra) oppure in rami alternativi mutuamente esclusivi.

¹I. Lanese, C. Guidi, F. Montesi, G. Zavattaro: *Bridging the Gap between Interaction- and Process-Oriented Choreographies*, SEFM 2008.

2.3.2 Analisi sulla coreografia di ACME Mobility

Connectedness per la sequenza. Tutte le composizioni sequenziali nella coreografia soddisfano la condizione: in ogni giuntura $C; C'$ almeno un partecipante è comune tra l'ultima interazione di C e la prima di C' . I casi più significativi sono:

- **return_vehicles:** $RS \rightarrow User$ seguito da **(scanning + booking):** $User \rightarrow RS$. $User$ è presente in entrambi i lati.
- **send_token:** $B \rightarrow RS$ seguito da **start_monitoring:** $RS \rightarrow FM$. RS è il receiver della prima e il sender della seconda.
- Blocco parallelo $FM-TS/BMS$ seguito da **ack_monitoring:** $FM \rightarrow RS$. FM è il join del parallelo e il sender dell'ack.
- Corpo del loop di monitoraggio (**vehicle_status_updated:** $RS \rightarrow User$) seguito dall'uscita (**end_ride:** $User \rightarrow RS$). $User$ è il receiver del corpo e il sender dell'uscita.

Connectedness per la scelta. Le scelte principali rispettano la condizione del sender comune:

- **Scenario A vs. B:** entrambi i rami iniziano con un messaggio di $User$ verso RS (**scanning** e **booking** rispettivamente). Il sender comune è $User$.
- **Esito bancario A1/A2 e B-ok/B-err:** entrambi i rami iniziano con un messaggio di B verso RS (**send_token** vs. **send_error**). Il sender comune è B .
- **Segnalazione danni vs. no-report:** rami iniziati rispettivamente da **send_report:** $User \rightarrow RS$ e **no_report_written:** $User \rightarrow RS$. Il sender comune è $User$.
- **Ricarica vs. no-ricarica:** entrambi i rami iniziano con un messaggio di RS verso S . Il sender comune è RS .
- **Annullamento esplicito (B1.1) vs. timeout (B1.2) vs. ritiro (B1.3):** B1.1 inizia con **cancel_reservation:** $User \rightarrow RS$, B1.3 con **scanning:** $User \rightarrow RS$ (sender $User$), ma B1.2 inizia con **timeout_message:** $RS \rightarrow User$ (sender RS). La condizione di sender comune non è soddisfatta in senso stretto. Questo riflette la natura ibrida del timeout: non è una scelta autonoma dell'utente ma un evento scatenato autonomamente da RS allo scadere del timer di prenotazione. In pratica RS non deve attendere un messaggio esterno per intraprendere il ramo B1.2, quindi l'ambiguità è risolta implicitamente dal controllo centralizzato dell'orchestratore; tuttavia si tratta di un raffinamento rispetto alla specifica formale pura.

Causality-safety. Non esistono nella coreografia due comunicazioni sulla stessa operazione tra gli stessi partecipanti che siano abilitate contemporaneamente:

- **block_money**: $RS \rightarrow B$ compare sia nello Scenario A sia nello Scenario B, ma i due scenari sono in alternativa (+), dunque non sono mai attivi insieme.
- **start_monitoring** e **stop_monitoring** tra RS e FM sono in sequenza rigorosa (il secondo segue il completamento del primo).
- Il corpo del loop di monitoraggio esegue **request_position** e **request_battery_status** a ogni iterazione; la semantica dell'operatore $*$ garantisce che ogni iterazione si completi prima che la successiva inizi, escludendo sovrapposizioni.

2.4 Proiezioni locali dei ruoli

Le proiezioni sono ottenute sostituendo ogni interazione $l : A \rightarrow B$ con:

- $1!B$ se il partecipante proiettato è A (invio),
- $1?A$ se è B (ricezione),
- 1 (skip) altrimenti.

Le proiezioni formali complete per tutti gli otto partecipanti sono disponibili nella cartella **deliverables/projections/** e consultabili nella raccolta degli artefatti indicata nella Sezione 1.4.

3 Modellazione dei processi BPMN

3.1 Diagramma di collaborazione documentativo

L'artefatto `general_diagram_doc.bpmn` rappresenta un diagramma di collaborazione BPMN che comprende otto *pool* espanse, corrispondenti ai partecipanti Rental Service, User, Bank, Station, Fleet Management, Tracking Service, Battery Monitoring Service e Maintenance Service. Tale diagramma ha finalità prettamente documentativa: descrive dettagliatamente i processi e ciò che accade al loro interno, ma non costituisce l'artefatto da eseguire sul motore di processo.

- **Rental Service:** svolge il ruolo di coordinatore della collaborazione. Gestisce le alternative di noleggio immediato e prenotazione breve, la pre-autorizzazione bancaria, l'avvio e la chiusura del noleggio e le attività successive alla riconsegna.
- **User:** rappresenta le interazioni dell'utente con il sistema, dalla consultazione dei veicoli alla scelta tra scansione e prenotazione, fino alla conclusione della corsa e all'eventuale segnalazione di un danno.
- **Bank:** gestisce il blocco cauzionale, i relativi esiti, lo sblocco o l'addebito della cauzione e il pagamento finale.
- **Station:** esegue lo sblocco e il blocco fisico del veicolo e ne gestisce l'eventuale ricarica dopo la riconsegna.
- **Fleet Management:** coordina in parallelo Tracking Service e Battery Monitoring Service, aggregandone le risposte per produrre gli aggiornamenti di telemetria richiesti durante il noleggio.
- **Tracking Service e Battery Monitoring Service:** forniscono rispettivamente posizione e stato della batteria e seguono il ciclo di avvio, interrogazione e arresto del monitoraggio.
- **Maintenance Service:** accoda i veicoli segnalati e, nel processo periodico di manutenzione, ne aggiorna lo stato comunicandolo al Rental Service.

La lettura complessiva del modello evidenzia tre macrofasi. La prima comprende la selezione del veicolo, la verifica della cauzione e l'eventuale prenotazione; la seconda attiva il veicolo e il monitoraggio, recupera posizione e batteria e restituisce gli aggiornamenti all'utente; la terza chiude il noleggio, regola pagamento e cauzione e instrada il veicolo verso ricarica o manutenzione quando necessario.

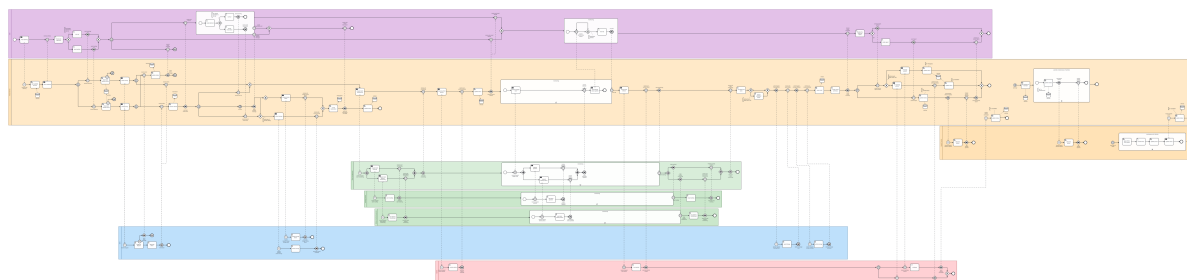


Figura 2: Diagramma di collaborazione BPMN documentativo del sistema ACME Mobility.

3.2 Processo di noleggio eseguibile

L'artefatto `rental_process.bpmn` descrive il processo eseguibile del Rental Service e traduce la vista documentativa della collaborazione in un flusso distribuibile su Camunda 8. In questo diagramma le interazioni esterne sono realizzate dai worker associati ai *service task*, mentre i messaggi correlati fanno avanzare l'istanza nei punti in cui il processo attende un'azione dell'utente o una risposta asincrona.

Il processo mantiene nel proprio stato le informazioni della prenotazione e del noleggio necessarie per la sessione. L'istanza viene creata esplicitamente dal Rental Service per l'utente e rimane attiva fino alla conclusione dell'operazione o alla gestione di un esito anticipato.

3.2.1 Lettura sequenziale del processo

Il flusso principale è composto dalle seguenti fasi.

Scelta dell'operazione

Dopo lo *start event*, un *event-based gateway* attende uno tra due messaggi alternativi: la scansione del veicolo avvia il noleggio immediato, mentre la richiesta di prenotazione attiva il ramo dedicato.

Preparazione e pre-autorizzazione

Il veicolo viene portato nello stato **BOOKED** e il task `blockMoney` richiede il blocco della cauzione. Nel ramo di prenotazione viene inoltre registrata la prenotazione e notificato l'esito all'utente; nel ramo immediato il processo può invece proseguire direttamente verso l'attivazione del noleggio.

Attivazione del noleggio

I due rami convergono prima dell'avvio del monitoraggio. Il processo richiede quindi l'attivazione del Fleet Management, lo sblocco fisico del veicolo alla Station e, dopo lo sblocco, aggiorna lo stato del veicolo a **RENTED**.

Monitoraggio attivo

Un sottoprocesso incorporato esegue ciclicamente il task `monitoringLogic`, notifica all'utente posizione e batteria e attende un secondo prima dell'itera-

zione successiva. Il messaggio `end_ride`, catturato da un evento di confine interrompente, termina il sottoprocesso e trasferisce il controllo alla chiusura.

Chiusura e regolazione economica

Il Rental Service richiede il blocco del veicolo e l'arresto del monitoraggio, addebita il prezzo della corsa, libera il blocco cauzionale, salva il noleggio e invia all'utente il riepilogo finale.

Gestione successiva alla riconsegna

Un nuovo *event-based gateway* attende la segnalazione di un danno oppure la conferma che non sia stato prodotto alcun report. Nel primo caso il veicolo passa a `MAINTENANCE` ed è inserito nella coda del servizio di manutenzione; nel secondo, il livello della batteria determina la richiesta di ricarica oppure il ritorno allo stato `AVAILABLE`. La sessione viene infine eliminata e il processo termina.

3.2.2 Prenotazione: eventi alternativi e compensazione

Dopo la registrazione della prenotazione, un *event-based gateway* mette il processo in attesa di tre eventi alternativi: la scansione del QR, la richiesta di annullamento o la scadenza di un timer (`bookingEndTime`). La scansione rimuove la prenotazione e fa convergere il flusso verso l'avvio del noleggio. Se arriva l'annullamento, un gateway esclusivo applica la regola temporale: con almeno cinque minuti residui il blocco cauzionale viene liberato, altrimenti viene addebitato. Anche la scadenza della prenotazione conduce all'addebito della cauzione.

Il modello usa un meccanismo di *compensazione* BPMN per gestire la casistica di annullamento della prenotazione. Raggiunto il *compensation end event*, vengono attivati gli handler delle attività completate: `unregisterBooking` annulla l'effetto di `registerBooking`, mentre `updateVehicleStatus` riporta il veicolo da `BOOKED` a `AVAILABLE`.

3.2.3 Esiti terminali: manutenzione e ricarica

Al termine del noleggio, il processo distingue tre esiti mutuamente esclusivi:

- **Manutenzione e ripristino della batteria.** Se l'utente segnala un danno, il veicolo passa a `MAINTENANCE` ed è inserito nella coda del Maintenance Service. Al completamento, il servizio comunicherà anche un livello di batteria pari al 100%; il processo asincrono `update_status` aggiorna quindi batteria e data dell'ultima manutenzione e riporta il veicolo a `AVAILABLE`. In questo caso, non viene richiesta una ricarica separata alla Station poiché si adotta l'interpretazione che la manutenzione comprenda anche il ripristino della batteria.
- **Ricarica senza manutenzione.** In assenza di una segnalazione di danno, se la batteria residua è inferiore al 15%, il task `requestRechargeVehicle` avvia la

ricarica asincrona presso la Station e il veicolo passa a **CHARGING**. La callback di completamento lo rende nuovamente **AVAILABLE** con la batteria aggiornata.

- **Ritorno diretto alla disponibilità.** Se non sono stati segnalati danni e la batteria è almeno al 15%, il task `noRechargeVehicle` conferma che la ricarica non è necessaria e il veicolo torna direttamente a **AVAILABLE**, conservando il livello residuo.

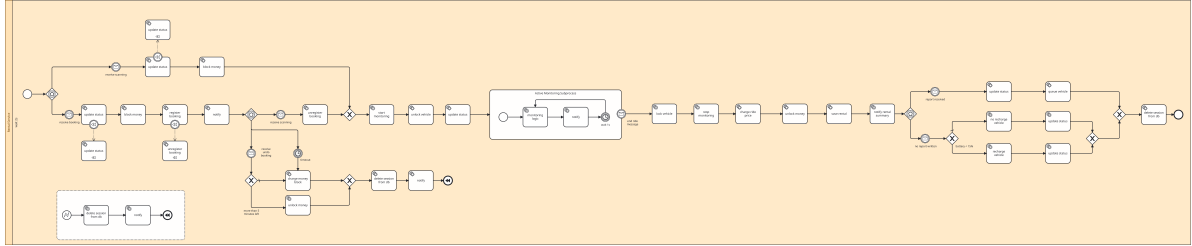


Figura 3: Processo BPMN eseguibile del noleggio (`rental_process.bpmn`).

3.3 Processi complementari al noleggio

Il processo principale di noleggio demanda a due processi eseguibili più piccoli le attività che non appartengono alla singola sessione utente: la selezione periodica dei veicoli da mantenere e l'applicazione degli aggiornamenti asincroni prodotti da manutenzione e ricarica.

3.3.1 Processo di manutenzione (`maintenance_process.bpmn`)

Il processo è avviato ogni notte a mezzanotte da un *timer start event*. Non esegue direttamente gli interventi, ma individua i veicoli interessati e li consegna al Maintenance Service:

1. il task `getVehiclesToMaintain` recupera i veicoli già in stato **MAINTENANCE** e quelli **AVAILABLE** la cui ultima manutenzione risale ad almeno 30 giorni prima;
2. un sottoprocesso multi-istanza sequenziale porta ciascun veicolo nello stato **MAINTENANCE**;
3. il task `queueVehicle` lo inserisce nella coda del Maintenance Service.

Il Maintenance Service elabora la coda ogni giorno alle 09:00, nel fuso orario Europe/Rome. La durata simulata di una manutenzione è:

$$t_{\text{manutenzione}} = 1 \text{ min} \times (1 + n_{\text{danni}})$$

Al termine, il servizio invia l'aggiornamento che riporta il veicolo a **AVAILABLE** con la batteria ripristinata al 100%, secondo la regola di dominio illustrata nella Sezione 3.2.3. La Figura 4 mostra la struttura compatta del processo periodico.

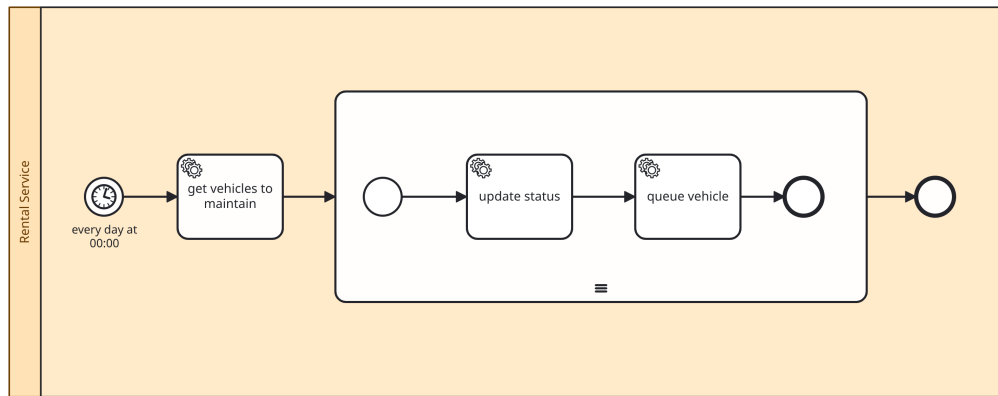


Figura 4: Processo BPMN di manutenzione preventiva periodica.

3.3.2 Processo di aggiornamento stato (`update_status.bpmn`)

Una nuova istanza del processo viene avviata dal messaggio `Message_updateStatus`. Riceve gli aggiornamenti prodotti sia dalla Station, al completamento di una ricarica, sia dal Maintenance Service, al termine di un intervento. Il task `updateVehicleStatus` applica in modo uniforme il nuovo stato e l'eventuale livello della batteria; nel passaggio da `MAINTENANCE` ad `AVAILABLE` aggiorna anche la data dell'ultima manutenzione.

In questo modo il completamento delle operazioni asincrone non prolunga né riattiva l'istanza del noleggio già conclusa. La Figura 5 rappresenta questo processo di raccordo.

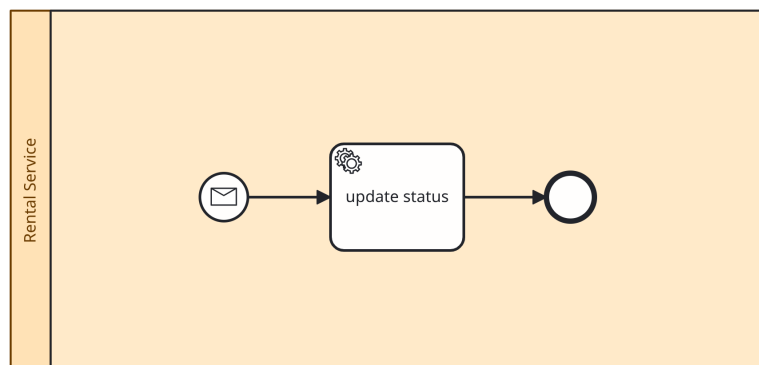


Figura 5: Processo BPMN di aggiornamento asincrono dello stato veicolo.

3.4 Coerenza fra BPMN documentativo e BPMN eseguibili

La specifica richiede che i processi BPMN eseguibili siano *consistenti* con il modello documentativo. Nel progetto, tale requisito è interpretato come **coerenza semantica** del processo, non come corrispondenza uno-a-uno fra ogni messaggio, task e meccanismo di comunicazione. I modelli conservano infatti gli stessi obiettivi di dominio, i partecipanti coinvolti, le principali alternative e l'ordine dei passaggi critici; il BPMN eseguibile introduce però adattamenti tecnici necessari al funzionamento del sistema.

Nel confronto, la coreografia e il diagramma di collaborazione costituiscono il **livello documentativo**: rappresentano partecipanti, responsabilità e ordine logico delle

interazioni. I BPMN eseguibili e i relativi worker costituiscono invece il **livello implementativo**: specificano task, timer, correlazioni e protocolli necessari all'esecuzione concreta.

Restano invariati, in particolare:

- l'attivazione del sottosistema di monitoraggio prima dello sblocco e l'avvio delle letture periodiche durante il noleggio;
- l'acquisizione periodica di posizione e livello della batteria;
- la consegna degli aggiornamenti all'utente durante il noleggio;
- l'interruzione del monitoraggio alla ricezione di `end_ride`.

3.4.1 Divergenza nel monitoraggio

La differenza principale riguarda l'attivazione del ciclo di monitoraggio. Nella coreografia ogni iterazione è rappresentata dall'interazione esplicita `request_update: User -> RS`; seguono `trigger_read: RS -> FM`, le letture parallele di tracking e batteria e infine la risposta verso l'utente. Questa sequenza rende visibili i ruoli e la relazione causale fra richiesta e aggiornamento, ma non coincide letteralmente con il comportamento dell'applicazione.

Nel BPMN eseguibile non è invece previsto il corrispondente messaggio ripetuto dell'utente. Dopo l'avvio del noleggio, il sottoprocesso di monitoraggio esegue autonomamente il job type `monitoringLogic`, attende un secondo e ripete il ciclo fino al messaggio `end_ride`. Il worker interroga il gateway di Fleet Management, che incapsula le letture dei servizi Tracking e Battery Monitoring; tali interazioni interne non compaiono quindi come task distinti nel processo eseguibile del Rental Service.

Anche la consegna del risultato adotta una semantica diversa. A ogni iterazione, il worker effettua una lettura **sincrona** verso Fleet Management e ne attende la risposta. Il task `notifyUser` pubblica poi la telemetria su Redis Pub/Sub e il Rental Service la inoltra al frontend attraverso un canale SSE persistente e unidirezionale. La consegna verso il browser è quindi **asincrona**: gli aggiornamenti vengono inviati non appena disponibili, senza una nuova richiesta dell'utente per ogni rilevazione. L'asincronicità introdotta da SSE riguarda dunque la distribuzione degli aggiornamenti al client, non la lettura della telemetria da parte del backend.

La divergenza è voluta: evita il polling del client, consente aggiornamenti continui e lascia al Fleet Management l'aggregazione della telemetria e al BPMS il controllo di frequenza, retry e interruzione. La coreografia e il BPMN documentativo descrivono quindi le responsabilità logiche, mentre il BPMN eseguibile le adatta all'architettura concreta. Il risultato osservabile resta lo stesso, pur senza una corrispondenza sintattica uno-a-uno fra gli artefatti.

4 Progettazione della SOA

4.1 Diagramma UML/TinySOA

La SOA di ACME Mobility è documentata tramite un diagramma UML con profilo **TinySOA** (disponibile nella cartella `deliverables/TinySOA/` del repository). Il modello include:

- **Vista dei servizi:** tutti i servizi con le rispettive interfacce, stereotipati come «**ServiceInterface**», e le operazioni atomiche che espongono, stereotipate come «**Capability**».
- **Vista delle dipendenze:** relazioni di esposizione («**Expose**») tra componenti e le capabilities che mettono a disposizione.
- **Vista di composizione:** i package sono organizzati con gli stereotipi «**Tasks**», «**Entities**» e «**Utilities**» per separare logicamente processi, entità di dominio e componenti di supporto.

Il profilo TinySOA applicato definisce i seguenti stereotipi:

- «**Capability**»: si applica a una `uml:Class` e rappresenta un'operazione atomica offerta da un servizio.
- «**ServiceInterface**»: si applica a una `uml:Interface` e modella il contratto esposto esternamente da un servizio.
- «**Expose**»: si applica a una `uml:Dependency` e modella la relazione con cui un componente rende disponibile una capability.
- «**Tasks**»: si applica a un `uml:Package` e raggruppa i processi e le operazioni orchestrate del sistema.
- «**Entities**»: si applica a un `uml:Package` e raggruppa le entità di dominio.
- «**Utilities**»: si applica a un `uml:Package` e raggruppa i componenti di supporto e infrastrutturali.

Il file `AcmeMobility.uml` contiene il modello EMF/UML completo.

Anche qui emerge una decomposizione delle responsabilità in tre gruppi funzionali:

- **Nucleo applicativo:** il Rental Service orchestra le *capabilities* del noleggio.
- **Servizi di dominio e di supporto:** Bank Service e Bank Account Registration gestiscono l'ambito finanziario; Stations e Maintenance governano il ciclo fisico del veicolo.
- **Sottosistema di monitoraggio:** il Fleet Management Gateway compone le funzionalità specializzate di Tracking e Battery.

I protocolli che realizzano le dipendenze fra questi servizi sono descritti nella sezione seguente.

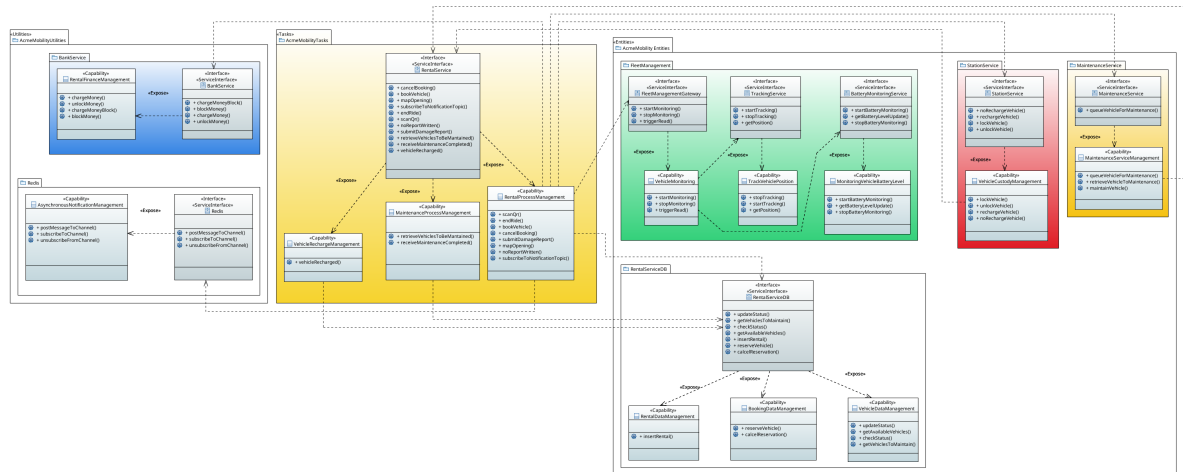


Figura 6: Diagramma UML con profilo TinySOA dell'architettura ACME Mobility.

4.2 Comunicazioni tra servizi

4.2.1 REST (HTTP/JSON)

La maggior parte delle comunicazioni sincrone inter-servizio avviene tramite API REST. Il Rental Service chiama:

- Stations Service (:8084) per sblocco, blocco e ricarica.
- Maintenance Service (:8085) per inserimento in coda manutenzione.
- Fleet Management Gateway (:8091) per avvio/stop monitoraggio e polling telemetria.

Il frontend Angular chiama:

- Rental Service (:8080) per tutte le operazioni applicative.
- Bank Account Registration (:8082) per la creazione account.

4.2.2 SOAP con Jolie

La comunicazione tra Rental Service (tramite worker Camunda) e Bank Service avviene via **SOAP**, come richiesto dalla specifica. Il Bank Service è implementato in **Jolie** ed espone un endpoint WSDL/SOAP sulla porta 8000.

Il worker Java costruisce la richiesta SOAP, la invia al Bank Service e gestisce la risposta (token di successo o fault di errore). Il profilo di interazione è *request/response* sincrono.

4.2.3 gRPC

La comunicazione tra Fleet Management Gateway e i due microservizi interni avviene tramite **gRPC**. I contratti `.proto` definiscono:

TrackingService

Avvio e arresto tramite `StartTracking` e `StopTracking`; lettura e aggiornamento tramite `GetPosition` e `UpdatePosition`.

BatteryService

Avvio e arresto tramite `StartBatteryMonitoring` e `StopBatteryMonitoring`; lettura e aggiornamento tramite `GetBatteryStatus` e `UpdateBattery`.

Le chiamate sono sincrone unarie. I microservizi espongono anche una porta HTTP per gli health check Docker.

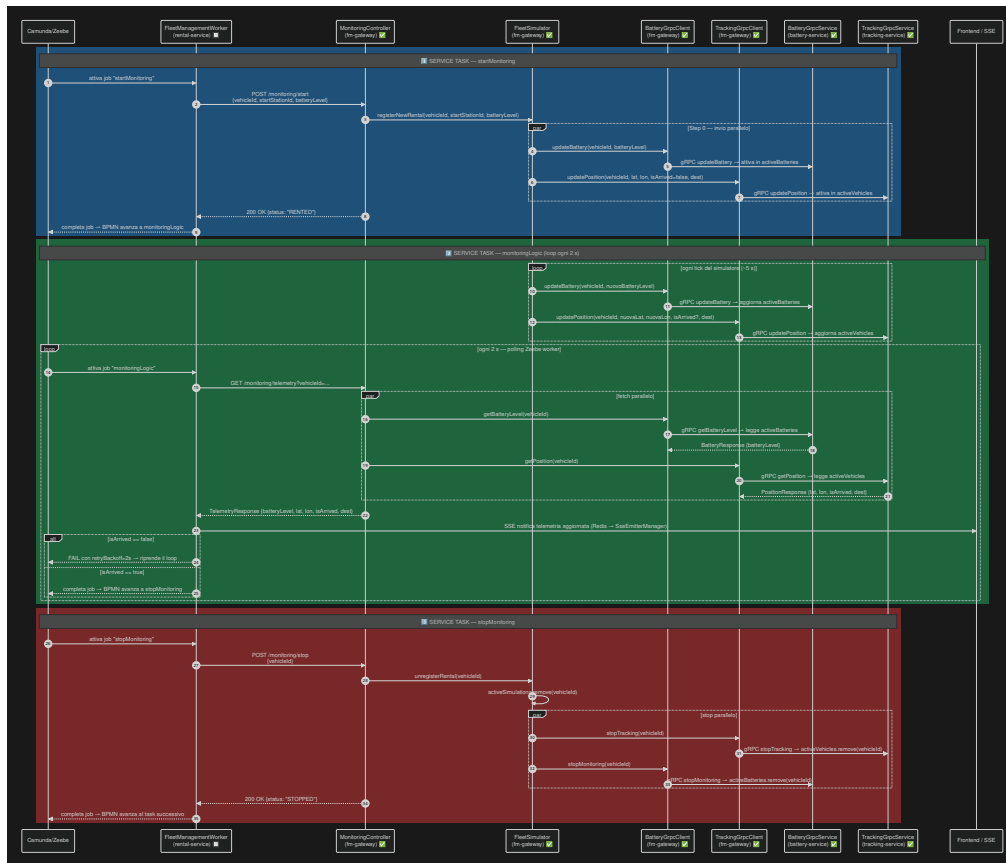


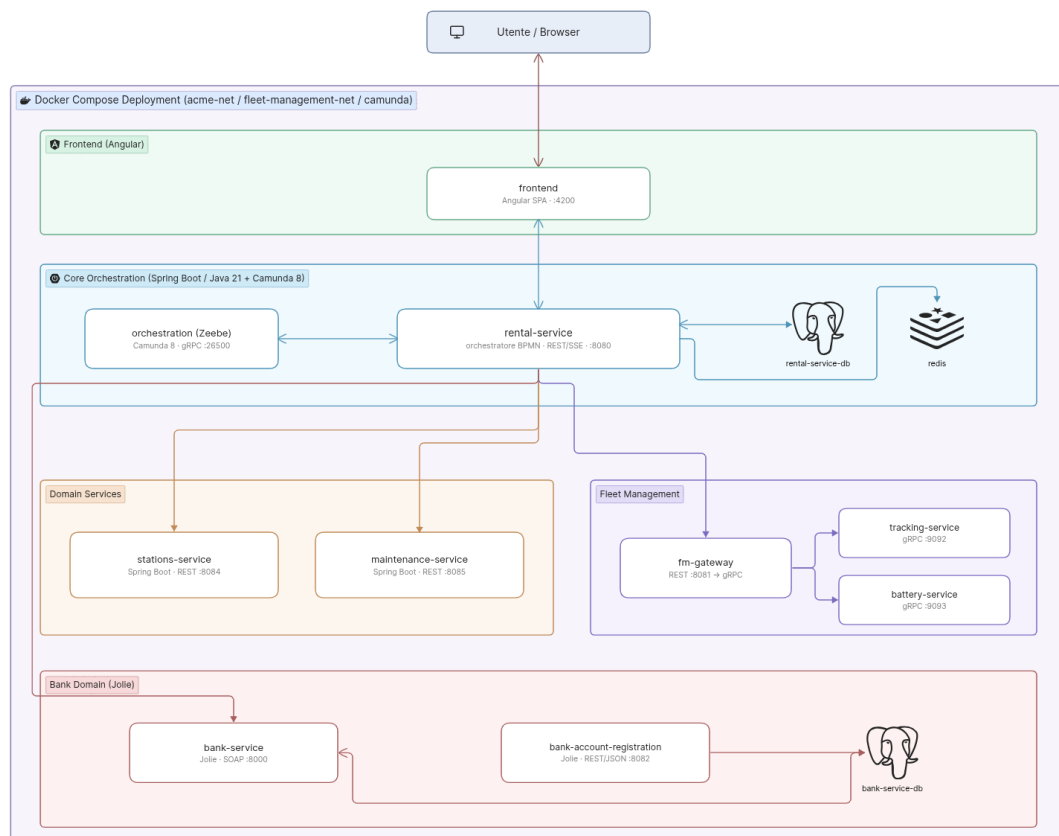
Figura 7: Schema architetturale del sottosistema Fleet Management: gateway REST e microservizi gRPC interni.

4.2.4 BPMS — Camunda/Zeebe

Il Rental Service comunica con Camunda 8 tramite il Zeebe Java client:

- **Deploy:** i file BPMN vengono deployati su Zeebe all'avvio.
- **Creazione istanza:** `createProcessInstance` per avviare un processo per ogni utente autenticato.
- **Correlazione eventi:** `publishMessage` per inviare comandi (scan, book, endRide, ecc.) all'istanza corretta, identificata tramite variabile di correlazione.
- **Job Workers:** un worker per ogni tipo di service task BPMN; ogni worker completa o fallisce il job al termine.

Redis Pub/Sub disaccoppia i worker Camunda dalla risposta SSE al frontend: i worker pubblicano su canali Redis, il controller SSE ascolta e invia gli eventi al browser.

ACME Mobility — Architettura a Microservizi (MSA-Project)

eraser

Figura 8: Schema dell'architettura generale del sistema con tutti i servizi e i protocolli di comunicazione.

5 Realizzazione e deployment

L'obiettivo di questa sezione è fornire gli strumenti concettuali e operativi necessari a comprendere l'organizzazione del sistema, le responsabilità dei singoli componenti e i flussi di coordinamento inter-servizio, offrendo una guida chiara a chi intenda analizzare l'architettura o estendere il codice sorgente senza doverne esaminare preventivamente ogni dettaglio implementativo. Nei paragrafi seguenti vengono quindi descritti i singoli servizi, le modalità di integrazione adottate, le procedure di containerizzazione e la fruizione dell'interfaccia utente.

5.1 Implementazione con Camunda e worker

5.1.1 Architettura modulare del Rental Service

Il Rental Service costituisce il fulcro dell'intera architettura applicativa. Sviluppato con Java 21 e Spring Boot 3, incapsula la gestione del dominio e funge da ponte tra le interazioni dell'utente, l'orchestrazione dei processi BPMN e i servizi di supporto. La struttura interna segue una separazione a strati ben definita:

- **Controllers:** gestiscono l'accesso HTTP esponendo le API REST pubbliche per l'autenticazione (`/api/auth/*`), la gestione operativa dei noleggi e delle prenotazioni (`/api/rentals/*`), lo streaming delle notifiche (`/api/notifications`) e gli endpoint riservati alle callback asincrone interne.
- **Services:** racchiudono la logica di business fondamentale, tra cui l'assegnazione automatica del veicolo con carica ottimale in fase di prenotazione, il calcolo delle tariffe di noleggio e l'applicazione delle penali su base temporale o di carica residua.
- **Repositories JPA:** interfacciano il database PostgreSQL dedicato (`rental-service-db`) per garantire la persistenza di utenti, veicoli, stazioni, prenotazioni, storico dei noleggi e sessioni di processo.
- **Client di integrazione:** incapsulano i contratti di rete verso i servizi esterni, offrendo interfacce tipizzate verso il client SOAP della Banca Jolie e i client REST verso Stations Service, Maintenance Service e Fleet Management Gateway.
- **Gestione delle sessioni di processo:** associa in modo univoco ciascun utente autenticato alla corrispondente istanza di processo Camunda, tracciandone la chiave di correlazione e la fase corrente.

5.1.2 Organizzazione dei Job Worker e orchestrazione Zeebe

L'avanzamento dei processi BPMN non è codificato mediante rigide sequenze di chiamate Java, ma è interamente demandato all'orchestratore Camunda 8. Il Rental Service

registra specifici *Job Worker* che interrogano costantemente il broker Zeebe per acquisire ed eseguire i *service task* di propria competenza:

BankWorker

Gestisce le interazioni SOAP con il servizio bancario. Richiede il blocco cauzionale di 10 €, salvando il token restituito nelle variabili di processo; qualora la banca segnali fondi insufficienti, solleva un *BPMN Error* che attiva il rispettivo boundary event di gestione dell'errore. Coordina inoltre l'addebito definitivo e lo sblocco cauzionale a fine corsa o in caso di annullamento.

FleetManagementWorker

Invia al gateway di Fleet Management la richiesta di avvio monitoraggio all'attivazione del noleggio e ne richiede l'arresto alla riconsegna. Durante la corsa, esegue periodicamente il job `monitoringLogic` per prelevare le coordinate aggiornate e il livello di batteria.

StationWorker

Inoltra i comandi REST di sblocco e blocco fisico alle singole stazioni, oltre a inoltrare le richieste di ricarica per i veicoli riconsegnati con batteria inferiore alla soglia minima.

VehicleWorker e RentalWorker

Aggiornano lo stato dei mezzi sul database di dominio (ad esempio da `BOOKED` a `RENTED` o `AVAILABLE`) e materializzano l'entità del noleggio calcolando l'importo consuntivo.

NotificationWorker

Intercetta gli eventi di avanzamento del processo e pubblica i relativi aggiornamenti sui canali Redis Pub/Sub, garantendone il recapito immediato all'interfaccia utente.

MaintenanceWorker

Invia i veicoli segnalati per danni al Maintenance Service e gestisce il processo batch notturno per i controlli periodici di revisione.

5.1.3 Navigazione guidata dal BPMN

L'applicazione frontend non determina autonomamente le transizioni di stato del noleggio, ma adotta un paradigma di navigazione strettamente guidato dall'orchestratore (*BPMN-driven routing*). Tramite l'endpoint `GET /api/rentals/resume`, il client riceve la fase corrente dell'istanza di processo associata all'utente autenticato:

BPMN state → Rental Service → `/api/rentals/resume` → Angular guard →
rotta canonica

In questo modo, anche a fronte di ricaricamenti di pagina, disconnessioni o navigazioni manuali dell'URL, le guardie di routing di Angular instradano sempre l'utente alla vista corrispondente all'effettivo stato del processo nel backend (mappa, prenotazione attiva, corsa in corso, riepilogo o segnalazione danni).

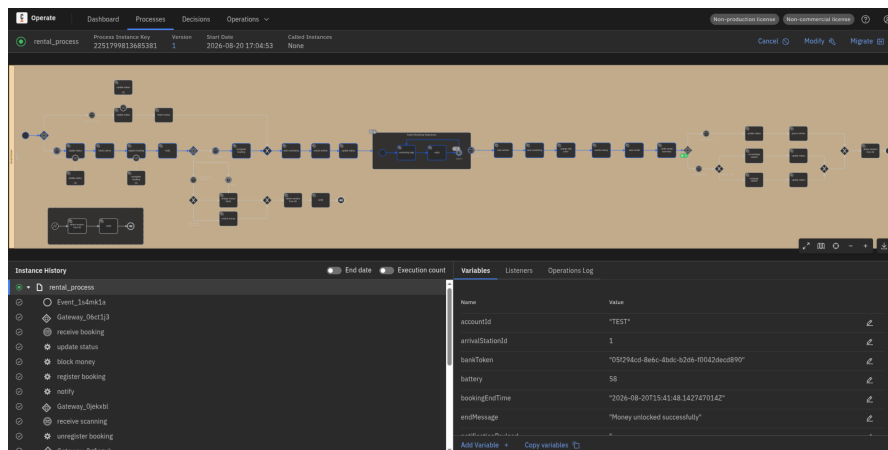


Figura 9: Camunda 8 Operate: istanza di processo attiva con variabili di correlazione e stato corrente dei service task.

5.2 Banca Jolie

5.2.1 Architettura a doppio servizio (SOAP e REST)

Il sottosistema bancario è interamente realizzato con il linguaggio orientato ai servizi **Jolie** ed è strutturato in due componenti distinti che condividono la medesima base di dati PostgreSQL (**bank-service-db**, porta 5433):

bank-service

Esponde un'interfaccia formale **SOAP/WSDL** sulla porta 8000. È il servizio deputato a interagire con l'orchestratore Camunda per l'esecuzione delle operazioni transazionali di blocco, addebito e sblocco cauzionale, nel rispetto dei vincoli architetturali di progetto.

bank-account-registration-service

Esponde un endpoint **REST** (POST /createAccount) sulla porta 8082. È disaccoppiato dal servizio SOAP principale per consentire al frontend Angular di registrare direttamente un conto bancario di prova con saldo iniziale di 100 € contestualmente all'iscrizione dell'utente, senza richiedere stack client SOAP nel browser.

5.2.2 Interfaccia SOAP e modello transazionale

Il contratto del **bank-service** definisce le operazioni necessarie a gestire l'intero ciclo finanziario:

- `blockMoney(accountId)`: verifica la disponibilità del conto e riserva l'importo cauzionale di 10€, generando un token univoco memorizzato nelle variabili di processo BPMN.
- `chargeMoney(accountId, token, amount)`: addebita la tariffa consuntiva di noleggio scalando il saldo totale del cliente e rilasciando il vincolo cauzionale.
- `unlockMoney(accountId, token)`: rilascia l'importo cauzionale precedentemente bloccato, ripristinando la piena disponibilità della somma.
- `chargeMoneyBlock(accountId, token)`: converte integralmente il blocco cauzionale in addebito definitivo in caso di mancato ritiro o di annullamento tardivo della prenotazione.

Qualora la richiesta di blocco o di addebito non possa essere soddisfatta per fondi insufficienti, il servizio Jolie restituisce un esito negativo strutturato che viene convertito dal worker Java in un errore di processo gestibile da Camunda tramite boundary events.

5.2.3 Gestione dello stato contabile e correlazione

La persistenza su PostgreSQL traccia lo stato contabile separando il saldo reale (`balance`) dall'ammontare temporaneamente vincolato (`blocked_amount`). Ogni sessione di noleggio è distinta e identificata lato banca tramite l'utilizzo di un correlation set. Esso correla un token generato dalla banca al momento del blocco della cauzione, ammesso che questa operazione vada a buon fine. Tutte le successive operazioni, alla fine di poter essere eseguite correttamente, devono essere correlate con il token precedentemente generato. L'esito economico complessivo (durata, tariffa consuntiva, eventuali penali e importo addebitato) viene calcolato e presentato all'utente nella schermata di riepilogo al termine della corsa, evitando la necessità di un'interrogazione continua del saldo verso il servizio bancario.

5.3 Fleet Management come microservizi

5.3.1 Decomposizione a microservizi e protocolli di comunicazione

Il sottosistema di Fleet Management è implementato come applicazione a microservizi autonoma, isolata all'interno della rete Docker dedicata `fleet-management-net`. La responsabilità del tracciamento è ripartita su tre componenti distinti:

fm-gateway

Costituisce il punto di accesso unico (API Gateway) verso l'esterno, esposto su porta 8091 con interfaccia REST. Coordina il ciclo di vita del monitoraggio, gestisce il motore di simulazione dei percorsi e interroga in parallelo i servizi interni.

tracking-service

Microservizio gRPC (porta interna 50051) che memorizza e aggiorna in modo concorrente le coordinate geografiche (latitudine e longitudine) dei veicoli attualmente in corsa.

battery-service

Microservizio gRPC (porta interna 50052) deputato al monitoraggio del livello percentuale di carica e del relativo consumo durante il tragitto.

La comunicazione interna tra Gateway, Tracking Service e Battery Service avviene esclusivamente tramite chiamate RPC sincrone definite mediante contratti Protocol Buffers (`.proto`), garantendo elevata efficienza e forte tipizzazione delle interfacce.

5.3.2 Simulazione del percorso con GraphHopper

Il modulo di simulazione interno al Gateway (**FleetSimulator**) riproduce in maniera realistica lo spostamento di ciascun veicolo noleggiato:

1. All'avvio del monitoraggio, il sistema seleziona una stazione di arrivo differente da quella di partenza all'interno del catalogo cittadino.
2. Il percorso effettivo lungo la rete stradale di Bologna viene calcolato interrogando il motore di routing **GraphHopper**; in caso di mancata risposta del motore cartografico, il sistema ricorre a una funzione di fallback basata sull'interpolazione lineare tra coordinate.
3. Il tragitto viene suddiviso in 20 segmenti discreti a velocità uniforme.
4. Dopo un tempo iniziale di preparazione pari a 15 secondi, uno scheduler aggiorna la posizione e decrementa la batteria ogni 3 secondi, invocando simultaneamente i metodi gRPC `UpdatePosition` e `UpdateBattery`.
5. Raggiunto l'ultimo segmento, il simulatore contrassegna il mezzo come giunto a destinazione, abilitando la procedura di riconsegna.

5.3.3 Aggregazione telemetrica e streaming SSE

Durante lo svolgimento della corsa, il Rental Service (tramite il worker `monitoringLogic`) interroga periodicamente il Fleet Management Gateway mediante la risorsa REST `GET /monitoring/telemetry/{vehicleId}`. Il Gateway aggrega i dati ricevuti via gRPC da Tracking e Battery Service e li restituisce al Rental Service all'interno di un unico DTO di risposta.

Ricevuto il DTO, il worker completa il job in Camunda; il motore di processo attiva quindi il task successivo (`notifyUser`), che pubblica il payload sul canale Redis Pub/Sub. Da qui, il gestore SSE del Rental Service inoltra l'evento al browser lungo la connessione

aperta, consentendo alla mappa Leaflet di aggiornare posizione e livello di batteria in tempo reale senza alcun polling dal client web.

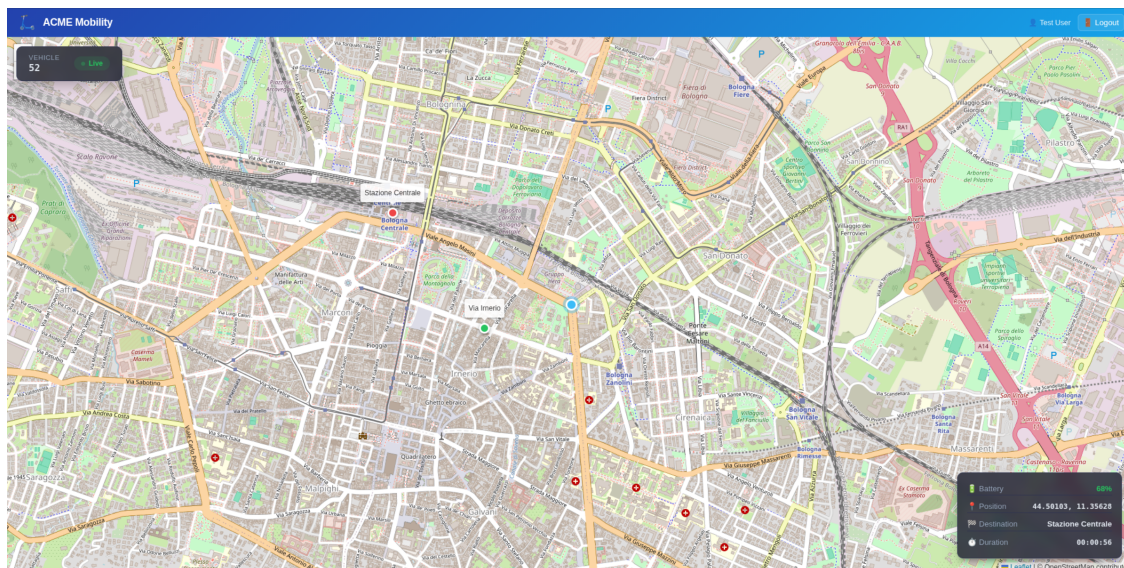


Figura 10: Schermata della corsa attiva con telemetria real-time (posizione sulla mappa, livello batteria e informazioni di corsa).

5.4 Stations Service

5.4.1 Catalogo e operazioni sui veicoli

Lo Stations Service, implementato con Spring Boot ed esposto sulla porta 8084, rappresenta il punto di accesso alle operazioni fisiche simulate delle stazioni. Il catalogo delle sette stazioni di Bologna è definito nella configurazione del servizio e reso disponibile tramite le API `GET /api/stations` e `GET /api/stations/{stationId}`.

Le operazioni di dominio sono esposte come risorse REST annidate sotto la stazione e il veicolo:

- `POST .../unlock` e `POST .../lock` simulano lo sblocco e il blocco fisico del mezzo;
- `POST .../recharge` avvia la ricarica fino al livello di batteria richiesto;
- `POST .../no-recharge` conferma che non è necessario avviare alcuna ricarica.

Il servizio verifica l'esistenza della stazione, la corrispondenza fra l'identificativo del veicolo nel percorso e quello nel corpo della richiesta e, per la ricarica, la validità dei livelli di batteria. Le operazioni di sblocco, blocco e mancata ricarica restituiscono la conferma dopo un ritardo fisico simulato configurabile, pari a 5 secondi nella configurazione corrente.

5.4.2 Ricarica asincrona e callback

La ricarica è modellata come operazione asincrona: l'endpoint restituisce HTTP 202 **Accepted**, mentre l'avanzamento viene eseguito su un executor dedicato. La durata simulata è proporzionale alla differenza fra il livello obiettivo e quello corrente:

$$t_{ricarica} = 1 \text{ s} \times (b_{obiettivo} - b_{corrente})$$

Al termine, lo Stations Service invia una callback interna al Rental Service. Quest'ultimo aggiorna lo stato persistito del veicolo attraverso il processo BPMN `update_status`, separando così l'avvio della ricarica dal suo completamento effettivo.

5.5 Maintenance Service

5.5.1 Coda ed elaborazione programmata

Il Maintenance Service, esposto sulla porta 8085, riceve dal Rental Service i veicoli danneggiati tramite `POST /api/maintenance/vehicles`. L'endpoint è riservato alle comunicazioni interne e richiede il token condiviso nell'header `X-Internal-Service-Token`.

Le richieste valide vengono inserite in una `LinkedBlockingQueue`; un insieme concorrente degli identificativi attivi impedisce che lo stesso veicolo sia accodato più volte. Alle 09:00, nel fuso orario `Europe/Rome`, uno scheduler avvia lo svuotamento asincrono della coda in ordine FIFO. La durata della simulazione dipende dal numero di danni segnalati:

$$t_{manutenzione} = 1 \text{ min} \times (1 + n_{danni})$$

Al completamento, il servizio notifica il Rental Service, che aggiorna il veicolo rendendolo nuovamente disponibile e ripristinandone il livello di batteria simulato.

5.5.2 Ricostruzione della coda

La coda è mantenuta in memoria, mentre lo stato autorevole dei veicoli risiede nel database del Rental Service. All'avvio, il Maintenance Service richiede quindi l'elenco dei veicoli ancora in manutenzione e ricostruisce la propria coda locale. Se il Rental Service non è ancora raggiungibile, il recupero viene ripetuto ogni 10 secondi fino al completamento. Questa scelta mantiene semplice il servizio di simulazione, evitando al contempo di perdere il lavoro pendente in seguito al riavvio del container.

5.6 Containerizzazione con Docker Compose

5.6.1 Struttura del Compose

Il file `docker-compose.yml` definisce 13 servizi containerizzati su tre reti Docker:

acme-net Rete principale: frontend, Rental Service, Stations, Maintenance, Bank e Redis.

camunda Rete isolata: Camunda/Zeebe e Rental Service.

fleet-management-net Rete isolata: FM Gateway, Tracking e Battery Service.

5.6.2 Health check e dipendenze

Ogni servizio definisce health check (`healthcheck:`) e dipendenze `depends_on` con condizione `service_healthy`, per garantire l'ordine di avvio corretto. In particolare:

- I database PostgreSQL devono essere `healthy` prima dei servizi applicativi.
- Camunda/Zeebe deve essere `healthy` prima del Rental Service.
- Il Rental Service deve essere `healthy` prima del frontend.

5.6.3 Volumi

I dati persistenti sono gestiti tramite volumi Docker nominati:

- `rental-db-data`: database PostgreSQL del Rental Service.
- `bank-db-data`: database PostgreSQL del Bank Service.
- `zeebe-data`, `camunda-data`: stato dell'orchestratore.

Sono disponibili anche volumi ri-creabili per cache Maven, `target`, `node_modules` e cache Angular.

5.7 Istruzioni minime di build e avvio

Il sistema può essere avviato localmente tramite Docker Compose, rispettando i prerequisiti riportati di seguito.

5.7.1 Requisiti

- Docker Engine con Docker Compose v2, almeno 8 GB RAM assegnati.
- Porte libere: 4200, 8000, 8080, 8082, 8083, 8084, 8085, 8091, 8092, 8093, 5432, 5433, 6379, 26500.
- Connessione Internet per il primo build.

5.7.2 Primo avvio

```
1 cd acme-mobility
2 docker compose up --build
```

Al completamento:

- Applicazione: `http://localhost:4200`

- Camunda UI: <http://localhost:8083> (credenziali: demo/demo)

5.7.3 Account demo

```
1 email:      user@acme.com
2 password:   password123
3 saldo:      100 EUR
```

5.7.4 Avvii successivi e pulizia

```
1 docker compose up          # avvio senza rebuild
2 docker compose down        # stop con preservazione dati
3 docker compose down -v     # reset completo (elimina volumi)
```

5.8 Interfaccia utente

Di seguito alcune schermate rappresentative dell'applicazione web.

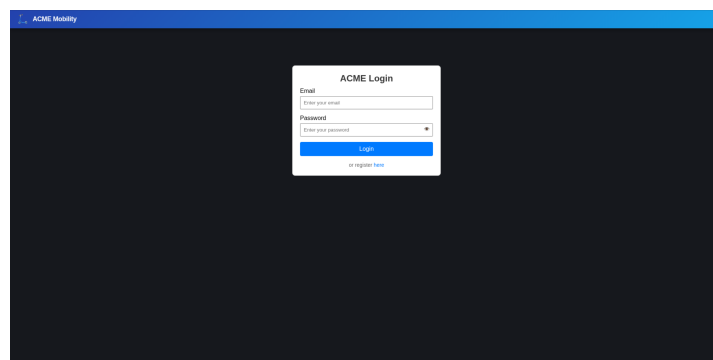


Figura 11: Schermata di login e registrazione utente.

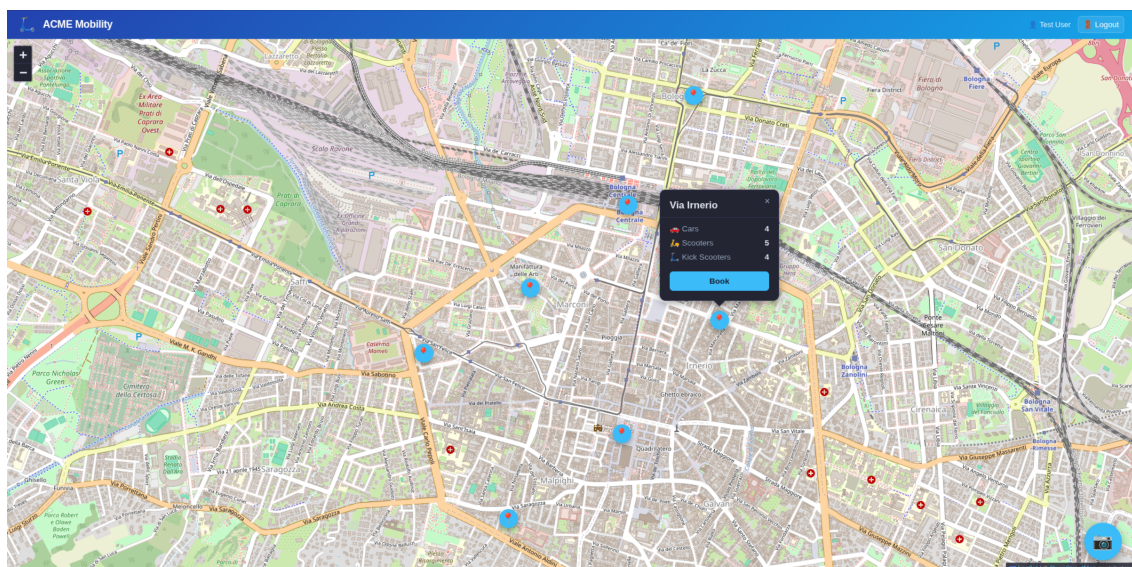


Figura 12: Vista panoramica della mappa con le stazioni di Bologna e i veicoli disponibili.

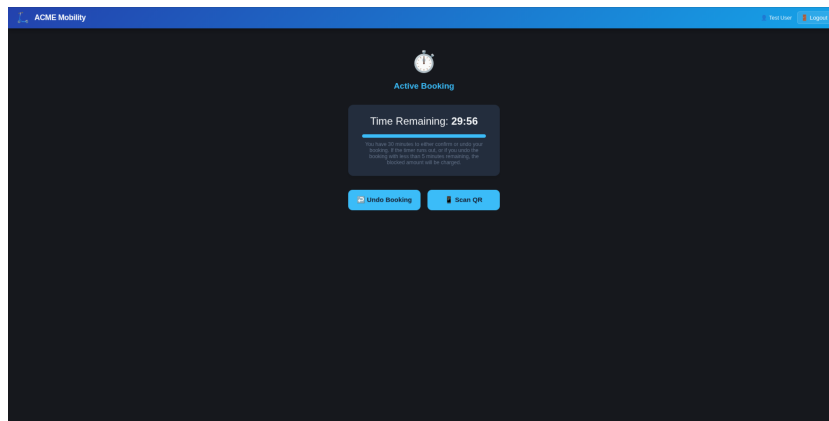


Figura 13: Schermata di prenotazione breve con countdown dei 30 minuti e pulsanti di ritiro/annullamento.

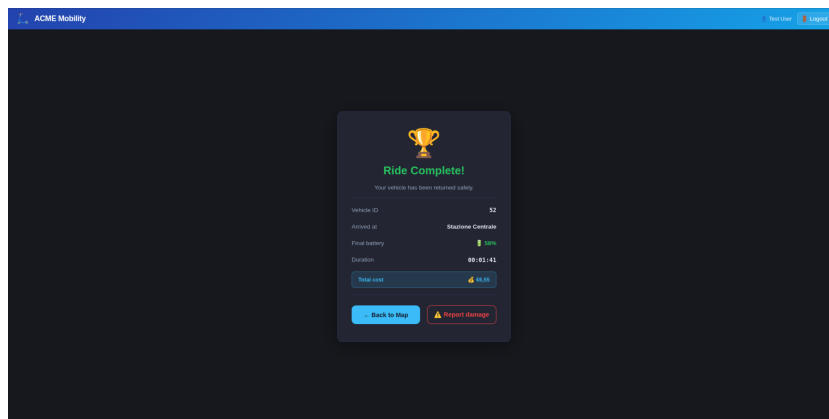


Figura 14: Riepilogo a fine corsa: durata, tariffa, penale batteria e saldo aggiornato.

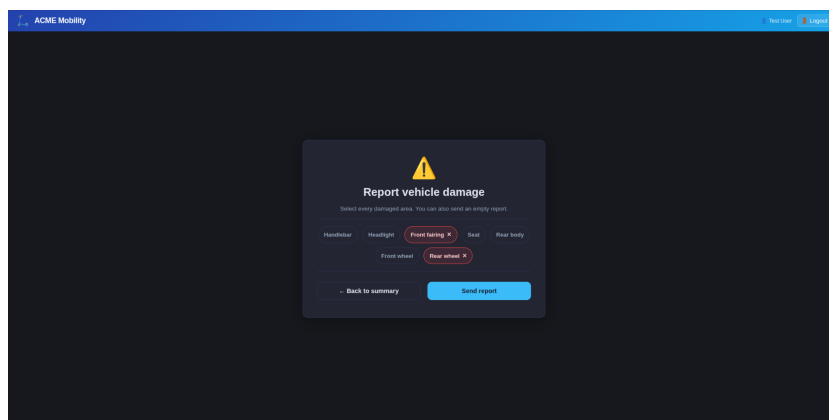


Figura 15: Schermata di segnalazione danni: selezione delle parti danneggiate per tipo di veicolo.

6 Conclusioni

6.1 Risultati raggiunti

Il progetto ha portato dalla specifica iniziale a un sistema distribuito eseguibile, mantenendo un collegamento esplicito fra modellazione, progettazione architetturale e scelte implementative. I risultati principali possono essere ricondotti a quattro aree.

Modellazione dei processi

Sono state definite la coreografia globale, le proiezioni dei ruoli e il modello BPMN documentativo. I flussi principali sono stati poi tradotti in processi eseguibili Camunda, documentando le divergenze introdotte consapevolmente nell'implementazione.

Orchestrazione del dominio

Il Rental Service governa tramite BPMN il noleggio immediato, la prenotazione breve, la pre-autorizzazione bancaria, il monitoraggio della corsa, la chiusura, la ricarica e l'avvio della manutenzione. Lo stato del processo guida inoltre la navigazione del frontend.

Integrazione di servizi eterogenei

L'architettura combina REST per le API applicative, SOAP per il servizio bancario Jolie, gRPC all'interno del Fleet Management e Redis Pub/Sub con SSE per gli aggiornamenti asincroni al browser. Stations e Maintenance completano il ciclo operativo attraverso simulazioni temporizzate e callback verso il Rental Service.

Esecuzione riproducibile

I componenti applicativi, i database e Camunda sono integrati tramite Docker Compose, con reti dedicate, health check, dipendenze di avvio e volumi per gli stati persistenti.

Gli artefatti aggiornati e le specifiche tecniche che documentano questi risultati sono raccolti nella GitHub Pages indicata nella Sezione 1.4.

6.2 Limiti e semplificazioni adottate

App mobile vs. applicazione web

La specifica prevede un'app mobile nativa. Il sistema realizzato è una *Progressive Web App* responsive con Angular, accessibile da browser desktop e mobile.

Scansione QR code

La scansione fisica è simulata tramite l'inserimento dell'ID del veicolo, in quanto l'accesso alla fotocamera non è disponibile in un contesto web generico.

Selezione veicolo nella prenotazione

La specifica descrive la prenotazione di un veicolo specifico. L'implementazione permette di prenotare per tipologia e stazione; il backend assegna il veicolo disponibile con il maggior livello di batteria.

Tariffa a tempo vs. tempo e distanza

La specifica prevede un costo basato su tempo e chilometri percorsi. Il sistema implementa una tariffa con quota fissa e quota a tempo, moltiplicata per un fattore 15x per rendere il pagamento osservabile durante una demo.

Storico noleggi

I noleggi completati sono persistiti nel database, ma il frontend non include una pagina di cronologia.

Coda di manutenzione in memoria

La coda FIFO del Maintenance Service è in memoria, ricostruita al riavvio dal Rental Service. Non è persistita su database.

Configurazione Camunda

La configurazione di Camunda/Zeebe è ottimizzata per sviluppo e dimostrazione. Lo storage secondario usa H2 file-based invece di PostgreSQL.

Sicurezza e resilienza

Non sono implementati HTTPS, OAuth2, circuit breaker o idempotenza bancaria. I rischi noti sono documentati in `ISSUES.md`.

Nonostante le semplificazioni, il sistema dimostra compiutamente i principi architetturali richiesti: coreografia e proiezione dei ruoli, orchestrazione BPMN con Camunda, integrazione SOAP con Jolie, decomposizione in microservizi con Fleet Management, comunicazione gRPC inter-microservizio, e deployment containerizzato con Docker Compose.