LiftLog Workout Tracker

1. Sommario

Il progetto LiftLog si inserisce nel dominio del fitness digitale, con l'obiettivo di supportare l'utente nella pianificazione e nel monitoraggio del proprio allenamento in palestra. L'applicazione consente di gestire schede di allenamento e logbook, suddividendo il percorso in mesocicli e microcicli, garantendo così un unico strumento centralizzato per la programmazione e la registrazione delle performance.

Le principali funzionalità sono:

- autenticazione light dell'utente;
- creazione e modifica di mesocicli e microcicli, con possibilità di duplicazione;
- gestione della scheda di allenamento, con impostazione di parametri e suddivisione della split settimanale;
- registrazione delle sedute svolte nel logbook, includendo parametri della giornata e note;
- visualizzazione dei grafici sul volume di lavoro per distretto muscolare.

Per la progettazione si è partiti dalla definizione del modello dati, per poi sviluppare parallelamente backend e frontend. Il back-end è stato realizzato in Node.js con Express.js, garantendo persistenza e validazione. La memorizzazione è gestita tramite file JSON. Il front-end è stato sviluppato con HTML5 e CSS3, sfruttando Flexbox e Grid per responsività. La parte dinamica è basata su JavaScript, Fetch API e manipolazione diretta del DOM. Per i grafici è stata adottata la libreria Chart.js.

2. Modello dei dati

L'applicazione si basa su un modello relazionale con le entità di seguito definite.

2.1 Utente (entità secondaria)

- id (chiave primaria, stringa univoca)
- username (stringa 3-20, univoco)
- password (stringa 6-30)

Relazione: 1 utente → N mesocicli

2.2 Mesociclo (entità principale)

- id (chiave primaria, stringa)
- ownerId (chiave esterna verso Utente, stringa)
- nome (stringa ≤ 30, obbligatorio)
- start (data ISO, obbligatoria)

Relazione: 1 mesociclo → N microcicli

2.3 Microciclo (entità secondaria)

- id (chiave primaria, stringa)
- mesoId (chiave esterna verso Mesociclo, stringa)
- nome (stringa ≤ 30, obbligatorio)

Relazione: 1 microciclo → 1 scheda, 1 logbook

2.4 Scheda di allenamento (entità secondaria)

- id (chiave primaria, stringa)
- microId (chiave esterna verso Microciclo, stringa)
- split (array di 14 stringhe ≤ 40)
- params: durata, densità, intensità_carico, intensità_perc, volume (stringhe ≤ 20)

Relazione: 1 scheda → N workoutPrev

2.5 Workout previsto (entità secondaria)

- id (chiave primaria, stringa)
- schedaId (chiave esterna verso Scheda, stringa)
- nomeSeduta (stringa ≤ 40)

Relazione: 1 workoutPrev → N eserciziPrev

2.6 Esercizio previsto (entità secondaria)

- id (chiave primaria, stringa)
- workoutPrevId (chiave esterna verso Workout previsto, stringa)
- esercizio (stringa ≤ 60), recupero (stringa ≤ 60)
- set1 - set5 (stringhe ≤ 60), tempo1 - tempo5 (stringhe ≤ 60)
- gm (valore enum tra petto, dorso, bicipiti, tricipiti, ecc.)

2.7 Logbook (entità secondaria)

- id (chiave primaria, stringa)
- microId (chiave esterna verso Microciclo, stringa)

Relazione: 1 logbook → N workoutEse

2.8 Workout eseguito (entità secondaria)

- id (chiave primaria, stringa univoca)
- logbookId (chiave esterna verso Logbook, stringa)
- nomeSeduta (stringa ≤ 40), data (data ISO)
- durata (stringa ≤ 40), peso (stringa ≤ 40), oreSonno (stringa ≤ 40), alimentazione (stringa ≤ 40), qualita (stringa ≤ 20), cardio (stringa ≤ 40), note (stringa ≤ 500)

Relazione: 1 workoutEse → N eserciziEse

2.9 Esercizio eseguito (entità secondaria)

- id (chiave primaria, stringa univoca)
- workoutEseId (chiave esterna verso Workout eseguito, stringa)
- esercizio (stringa ≤ 60)
- set1 - set5 (stringhe ≤ 60), carico1 - carico5 (stringhe ≤ 60)

Nell'implementazione su file JSON è stato introdotto l'attributo ownerId in tutte le entità, al fine di semplificare i controlli di autorizzazione, i filtri per utente e le operazioni di duplicazione. Questa scelta aumenta la ridondanza ma riduce la complessità del codice.

3. Implementazione del backend

Il back-end è stato realizzato con Node.js e Express.js, organizzato come server RESTful.

3.1 Architettura del server

Il server è stato implementato interamente nel file server.js, situato nella cartella backend. L'applicazione utilizza Node.js con il framework Express.js per la gestione delle richieste HTTP. La cartella data contiene i file JSON per la persistenza. Gli array sono caricati in memoria all'avvio tramite la funzione loadAll(), che sfrutta safeLoad per leggere in sicurezza i file e save per scrivere le modifiche. Inoltre, funzioni helper aiutano la gestione degli identificativi (nextId) e la validazione date (isValidISODate).

Sono stati adottati diversi middleware:

- app.use(express.json()): middleware built-in per interpretare le richieste con corpo in formato JSON.
- middleware di autenticazione light: verifica la presenza del token nell'header X-Auth-Token. Se manca restituisce 401 Unauthorized, se non corrisponde ad alcun utente restituisce 403 Forbidden. In caso positivo aggiunge l'identificativo dell'utente alla richiesta, consentendo così di associarla all'utente corretto e proseguire.
- express.static(...): utilizzato per servire i file statici del frontend.

Infine, è stata definita una route di servizio statico app.get('/'), che restituisce la pagina iniziale index.html della home.

3.2 Endpoint REST

Il server espone endpoint REST organizzati per ogni risorsa, basati sui metodi GET, POST, PUT e DELETE. In questa applicazione, la route PUT funziona solo come aggiornamento: riceve e restituisce l'oggetto completo (inclusi i campi non modificati), non creando la risorsa se non esiste. Quest'ultima funzione rimane separata e gestita tramite POST, con ID generato lato server. Sono presenti anche alcune rotte aggiuntive: una per la duplicazione dei microcicli e due per il calcolo delle serie allenanti dei grafici del volume per gruppo muscolare. Le rotte per ogni risorsa sono descritte di seguito.

3.2.1 Autenticazione e utente corrente

- `POST /api/register`
  Scopo: registrare un nuovo utente. Richiesta: invio di username e password. Validazioni: username stringa di 3 - 20 caratteri e non già in uso; password stringa di 6 - 30 caratteri. Risposte: 201 in caso di successo, 400 se i vincoli non sono rispettati, 409 se l'username è già presente.
- `POST /api/login`
  Scopo: autenticare un utente e restituire un token. Richiesta: invio di username e password. Risposte: 200 se le credenziali sono corrette, 401 se non valide.
- `GET /api/users/current`
  Scopo: ottenere i dati dell'utente autenticato. Richiesta: invio del token di autenticazione. Risposte: 200 con i dati utente, 404 se non trovato.
- `DELETE /api/users/current`
  Scopo: eliminare l'utente e tutti i dati a lui collegati. Richiesta: invio del token di autenticazione. Risposte: 200 se eliminato, 404 se non trovato.

3.2.2 Mesocicli

- `GET /api/mesocicli`
  Scopo: ottenere la lista dei mesocicli dell'utente. Validazioni: filtro per ownerId. Risposte: 200 con array di mesocicli.
- `GET /api/mesocicli/:id`
  Scopo: ottenere i dettagli di un singolo mesociclo. Richiesta: parametro id nella URL. Validazioni: esistenza e appartenenza all'utente. Risposte: 200 con oggetto trovato, 404 se non esiste.
- `POST /api/mesocicli`
  Scopo: creare un nuovo mesociclo. Richiesta: corpo con nome (stringa ≤ 30, obbligatorio) e start (stringa ISO YYYY-MM-DD). Validazioni: lunghezza nome, formato data. Risposte: 201 con oggetto creato, 400 se vincoli non rispettati.
- `PUT /api/mesocicli/:id`
  Scopo: aggiornare un mesociclo esistente. Richiesta: corpo con nome e start validi. Validazioni: esistenza mesociclo, appartenenza all'utente, lunghezza nome, formato data. Risposte: 200 con oggetto aggiornato, 400 se dati non validi, 404 se non trovato.
- `DELETE /api/mesocicli/:id`
  Scopo: eliminare un mesociclo e tutte le entità collegate. Richiesta: parametro id nella URL. Validazioni: esistenza e appartenenza all'utente. Risposte: 200 se eliminato, 404 se non trovato.

3.2.3 Microcicli

- `GET /api/microcicli/:mesoId`
  Scopo: elenca i microcicli di un mesociclo. Richiesta: parametro :mesoId. Validazioni: controllo appartenenza al meso e all'utente. Risposte: 200 con lista micro.
- `POST /api/microcicli`
  Scopo: crea un nuovo microciclo. Richiesta: body {mesoId}. Validazioni: mesoId obbligatorio ed esistenza meso. Risposte: 201 con micro creato, 400 se mancante, 404 se meso non trovato.
- `POST /api/microcicli/:id/copies`
  Scopo: duplica un microciclo con scheda, workout, esercizi e logbook. Richiesta: parametro :id, body { mesoId }. Validazioni: esistenza micro sorgente e meso target. Risposte: 201 con micro duplicato, 404 se entità mancanti.
- `DELETE /api/microcicli/:id`
  Scopo: elimina un microciclo e i dati collegati. Richiesta: parametro :id. Validazioni: controllo esistenza e appartenenza. Risposte: 200 con conferma eliminazione, 404 se non trovato.
- `GET /api/microcicli/:mesoId/stats/serie-gm`
- Scopo: restituisce per ogni microciclo del meso il numero di serie per un gruppo muscolare. Richiesta: parametro :mesoId, query gm. Validazioni: gm obbligatorio e valido, meso esistente e appartenente all'utente. Risposte: 200, 400 se parametri invalidi, 404 se meso non trovato.

3.2.4 Scheda di allenamento

- `GET /api/schede/:microId`
  Scopo: ottenere la scheda collegata a un microciclo. Richiesta: :microId. Validazioni: esistenza micro e scheda dell'utente. Risposte: 200 con oggetto scheda, 404 se micro o scheda non trovati.
- `POST /api/schede/:microId`
  Scopo: creare la scheda per un microciclo. Richiesta: :microId. Validazioni: micro esistente, assenza di scheda già collegata. Risposte: 201 con scheda inizializzata, 404 se micro non trovato, 409 se scheda già esistente.
- `PUT /api/schede/:id`
  Scopo: aggiornare una scheda esistente. Richiesta: body con split (array di 14 stringhe ≤ 40) e params oggetto con chiavi obbligatorie (stringhe ≤ 20). Validazioni: scheda esistente, corretto formato dei dati. Risposte: 200 con scheda aggiornata, 404 se scheda non trovata, 400 se split/params non validi.
- `GET /api/schede/:schedaId/stats/serie-gm`
  Scopo: conteggio serie per gruppo muscolare sulla scheda. Richiesta: :schedaId. Validazioni: scheda esistente e appartenenza all'utente. Risposte: 200, 404 se scheda non trovata, 403 se non appartiene all'utente.

3.2.5 Workout previsti

- `GET /api/workoutsPrev/:schedaId`
  Scopo: ottenere i workout previsti collegati a una scheda. Richiesta: :schedaId. Validazioni: esistenza della scheda e appartenenza all'utente. Risposte: 200 con lista di workout, 404 se scheda non trovata.
- `POST /api/workoutsPrev/:schedaId`
  Scopo: creare un nuovo workout previsto per una scheda. Richiesta: :schedaId. Validazioni: esistenza e appartenenza della scheda. Risposte: 201 con nuovo workout creato, 404 se scheda non trovata.
- `PUT /api/workoutsPrev/:id`
  Scopo: aggiornare un workout previsto. Richiesta: body con nomeSeduta (stringa ≤ 40, obbligatoria). Validazioni: esistenza workout, scheda collegata dell'utente, corretto formato del campo. Risposte: 200 con workout aggiornato, 404 se workout non trovato, 403 se scheda non appartiene all'utente, 400 se nomeSeduta non valido.
- `DELETE /api/workoutsPrev/:id`
  Scopo: eliminare un workout previsto e i relativi esercizi collegati. Richiesta: :id. Validazioni: esistenza workout e appartenenza della scheda. Risposte: 200 con messaggio di conferma, 404 se workout non trovato, 403 se non autorizzato.

3.2.6 Esercizi previsti

- `GET /api/eserciziPrev/:workoutPrevId`
  Scopo: ottenere le righe esercizi previsti di un workout previsto. Richiesta: :workoutPrevId. Validazioni: esistenza workout previsto; scheda collegata appartenente all'utente. Risposte: 200 lista esercizi; 404 se seduta non trovata; 403 se non autorizzato.
- `POST /api/eserciziPrev/:workoutPrevId`
  Scopo: creare una nuova riga esercizio. Richiesta: :workoutPrevId. Validazioni: esistenza workout previsto; scheda collegata appartenente all'utente. Risposte: 201 riga esercizio creato.
- `PUT /api/eserciziPrev/:id`
  Scopo: aggiornare una riga esercizio. Richiesta: body con campi obbligatori esercizio, recupero, set1 - set5, tempo1 - tempo5, gm. Validazioni: riga esistente; workout e scheda dell'utente; gm ∈ {/, petto, dorso, …}; ogni campo ≤ 60; tutti i campi presenti. Risposte: 200 aggiornato; 404 se riga esercizio non trovato; 403 se non autorizzato; 400 input non valido.
- `DELETE /api/eserciziPrev/:id`
  Scopo: eliminare una riga esercizio. Richiesta: :id. Validazioni: riga esistente; scheda collegata appartenente all'utente. Risposte: 200; 404; 403.

3.2.7 Logbook

- `GET /api/logbooks/:microId`
  Scopo: ottenere il logbook del microciclo. Richiesta: :microId. Validazioni: esistenza del microciclo dell'utente; presenza del logbook collegato. Risposte: 200; 404 se microciclo non trovato o logbook assente.
- `POST /api/logbooks/:microId`
  Scopo: creare il logbook del microciclo. Richiesta: :microId. Validazioni: microciclo dell'utente esistente; unicità del logbook per quel microciclo. Risposte: 201 logbook creato; 404 microciclo non trovato; 409 logbook esistente.

3.2.8 Workout eseguito

- `GET /api/workoutsEse/:logId`
  Scopo: elencare i workout eseguiti del logbook. Richiesta: :logId. Validazioni: esistenza del logbook utente. Risposte: 200 lista; 404 logbook inesistente o non autorizzato.
- `POST /api/workoutsEse/:logId`
  Scopo: creare un nuovo workout eseguito per il logbook. Richiesta: :logId. Validazioni: logbook dell'utente esistente. Risposte: 201 oggetto creato; 404 logbook inesistente o non autorizzato.
- `PUT /api/workoutsEse/:id`
  Scopo: aggiornare un workout eseguito. Richiesta: :id + body completo con i vari parametri. Validazioni: workout esistente; appartenenza al logbook dell'utente; data ISO o vuota; lunghezze testi. Risposte: 200 oggetto aggiornato; 400 validazione; 403 non autorizzato; 404 workout non trovato.
- `DELETE /api/workoutsEse/:id`
  Scopo: eliminare un workout eseguito ed esercizi collegati. Richiesta: :id. Validazioni: workout esistente; appartenenza al logbook dell'utente. Risposte: 200 eliminato; 403 non autorizzato; 404 workout non trovato.

3.2.9 Esercizi eseguiti

- `GET /api/eserciziEse/:workoutEseId`
  Scopo: elencare le righe esercizio del workout eseguito; Richiesta: :workoutEseId; Validazioni: workout eseguito esistente; logbook associato appartenente all'utente; Risposte: 200, 403, 404.
- `POST /api/eserciziEse/:workoutEseId`
  Scopo: creare una nuova riga esercizio per il workout eseguito; Richiesta: :workoutEseId; Validazioni: workout eseguito esistente; logbook associato dell'utente; Risposte: 201, 403, 404.
- `PUT /api/eserciziEse/:id`
  Scopo: aggiornare una riga esercizio; Richiesta: :id + body completo con esercizio, set1 - set5, carico1 - carico5; Validazioni: riga esistente; appartenenza al logbook dell'utente; tutti i campi presenti (anche vuoti) e ciascuno ≤ 60; Risposte: 200, 400, 403, 404.
- `DELETE /api/eserciziEse/:id`
  Scopo: eliminare una riga esercizio; Richiesta: :id; Validazioni: riga esistente; appartenenza al logbook dell'utente; Risposte: 200, 403, 404.

4. Implementazione del frontend

4.1 Viste e organizzazione dei file

La parte front-end è organizzata in cartelle corrispondenti alle viste principali: home (info + pulsante "inizia ora"), dashboard/mesocicli (lista mesocicli con creazione e modifica), dashboard/microcicli (lista microcicli con modale per creazione/duplicazione e grafico delle serie per gruppo muscolare tra microcicli), dashboard/dettaglio (hub con accesso a scheda e logbook), dashboard/scheda_allenamento (split settimanale, parametri, sedute previste, grafico serie per gruppo muscolare nel microciclo) e dashboard/logbook (parametri giornalieri, sedute eseguite, note). Ogni cartella contiene index.html, style.css, script.js, mentre funzionalità comuni risiedono in /shared/ con style.css per palette/layout e auth.js per la gestione della modale di autenticazione e helper comuni. La cartella assets contiene le immagini e il font utilizzato.

4.2 Layout

Il layout delle pagine è costruito con HTML5 e CSS3, facendo uso di Flexbox e CSS Grid per gestire allineamenti e distribuzione degli elementi. L'header e il footer sono mantenuti tra le viste, mentre il contenuto centrale varia in base alla pagina visualizzata. Flexbox viene usato per la disposizione orizzontale di elementi come l'header, la navigazione e i pulsanti di azione. Le griglie sono adottate per strutturare le sezioni con i parametri e per affiancare dei blocchi di contenuto. Per la gestione dei workout e dei relativi esercizi sono state impiegate tabelle HTML, rese scorrevoli e adattive tramite wrapper con overflow. Sono previste media query per dispositivi sotto determinati px, che trasformano layout affiancati in blocchi verticali, rendendo l'interfaccia responsiva e usabile anche su schermi ridotti.

4.3 Autenticazione

L'autenticazione è gestita da auth.js tramite modale con login, registrazione e profilo. Lo stato è salvato in localStorage (authToken, username), che aggiunge la classe authenticated al body per mostrare/nascondere le tab corrette. Il login e la registrazione inviano richieste fetch agli endpoint dedicati, con controllo live della conferma password e auto-login in caso di successo; gli errori sono segnalati con alert. Dal profilo è possibile fare logout (rimozione token e redirect alla home) o eliminare l'account (previa conferma) con cancellazione dei dati.

4.4 Fetch API

La comunicazione con il backend avviene tramite Fetch API, consentendo richieste http asincrone. Ogni chiamata scambia dati in formato JSON, con Content-Type: application/json per le richieste che includono un body. Lo schema tipico prevede il controllo di r.ok, la gestione dei codici di stato (401/403, 404, 409) e la conversione della risposta in oggetto JavaScript tramite r.json().

4.5 Aggiornamento dinamico del DOM

Le pagine non si basano su template statici ma vengono generate dinamicamente a partire dai dati ricevuti. Gli elementi sono creati con document.createElement, popolati con i valori correnti e collegati a listener per gestire le azioni dell'utente. L'aggiunta o rimozione di un esercizio aggiorna subito la tabella, i logbook e le schede sono renderizzati dagli array di oggetti, le modali funzionano tramite il toggle di una classe CSS e la navigazione tra viste sfrutta query string che propagano id e nomi di mesocicli o microcicli. Per evitare eccessivi aggiornamenti e traffico verso il server, le operazioni come i salvataggi automatici usano un meccanismo di debounce, che ritarda l'invio fino a quando l'utente non interrompe l'attività per un breve intervallo (300ms).

4.6 Librerie esterne

Per la rappresentazione grafica dei volumi di allenamento è stata adottata la libreria Chart.js, integrata tramite server esterno (CDN). Sono stati sviluppati grafici a barre che mostrano il numero totale di serie per ciascun gruppo muscolare, sia a livello di più microcicli sia all'interno di ognuno di essi. I dati vengono ottenuti dalle due API dedicate alle statistiche e passati alla funzione new Chart() per l'inizializzazione. I grafici vengono aggiornati in tempo reale quando l'utente cambia i filtri o modifica i dati, sfruttando il metodo chart.update().

4.7 Feedback utente e validazioni lato client

Sono stati integrati sia meccanismi di feedback sia validazioni per migliorare l'esperienza utente e ridurre errori. I form sfruttano le validazioni native di HTML5 con vincoli su lunghezza, obbligatorietà e messaggi contestuali, estese da controlli personalizzati che rispecchiano le regole definite nel back-end: ad esempio username e password sono verificati già in input, la conferma password è controllata in tempo reale e i valori sono normalizzati con trim() per rimuovere spazi indesiderati. In caso di errori provenienti dal server o di operazioni non permesse, come tentare di duplicare un microciclo inesistente, vengono mostrati alert chiari. Per azioni distruttive, come l'eliminazione di una seduta o dell'intero account, è invece mostrato un messaggio di conferma. Questo livello di controlli, unito a pulsanti distinti per creazione, modifica ed eliminazione, garantisce un'interazione trasparente, chiara e robusta.

5. Conclusioni

Il progetto ha dimostrato come, partendo solamente dalle basi fornite dal corso, sia possibile realizzare un'applicazione completa e di buona qualità, capace di rispondere alle esigenze reali degli utenti del dominio scelto. La progettazione ha richiesto di affrontare alcune difficoltà: la definizione delle rotte di duplicazione dei microcicli, la gestione corretta delle rotte PUT (limitandole all'aggiornamento senza creare nuove risorse), la distinzione tra aggiornamenti completi e parziali in assenza del metodo PATCH, e la strutturazione delle funzioni di generazione delle sedute sia per schede che per logbook.

L'applicazione, pur sviluppata in ambito didattico, è già utilizzabile da atleti come strumento reale. Sono però possibili diversi miglioramenti, come una maggiore ottimizzazione dell'usabilità su dispositivi mobili, l'adozione di un database relazionale SQL al posto dei file JSON, e l'aggiunta di ulteriori funzioni di analisi, ad esempio una heatmap per valutare la qualità degli allenamenti annuali o il tracciamento del peso corporeo nel tempo.

Un'ulteriore prospettiva è l'estensione in ambito coaching: l'app potrebbe evolvere in una piattaforma per allenatori, con una dashboard centralizzata da cui accedere ai profili degli atleti per assegnare schede personalizzate e monitorare le loro performance nei rispettivi logbook.