Progetto di Sistemi Web - AA 2025/2026

Specifiche Progetto

1. Informazioni generali

- **Tipologia prova:** Progetto individuale full-stack (frontend + backend)
- **Tecnologie principali:**
    - **Frontend:** Angular, TypeScript
    - **Backend:** Ruby on Rails (in modalità API)
    - **Autenticazione:** OpenID Connect / OAuth2 (preferibile) oppure login con credenziali gestito dal backend

Lo studente parte da un'applicazione Angular di "shop online" sviluppata a lezione/lab e la completa trasformandola in una web app full-stack con:

- backend reale in Ruby on Rails,
- carrello persistente,
- autenticazione reale (non mockata),
- almeno una funzionalità avanzata a scelta.

2. Obiettivi formativi

Al termine del progetto lo studente dovrà dimostrare di saper:

1. Progettare e sviluppare un **backend REST** in Ruby on Rails, con modello dati coerente, validazioni e gestione degli errori.
2. Integrare un frontend Angular con un backend reale, sostituendo mock/fake con chiamate HTTP verso API REST.
3. Implementare un **flusso di autenticazione basilare** e proteggere risorse lato backend e frontend.
4. Gestire uno **stato applicativo persistente** (carrello, ordini) sincronizzato tra frontend e backend.
5. Curare aspetti di **qualità del codice**, organizzazione del progetto, UX di base e minima accessibilità.
6. Documentare l'applicazione (README, descrizione API e scelte architetturali).

3. Materiale di partenza

Si assume come base un'applicazione Angular di shop online con:

- catalogo prodotti (lista) e pagina di dettaglio prodotto;
- carrello gestito lato frontend (mock/in-memory/localStorage);
- pagina checkout con Reactive Forms e validazioni client;
- servizi HTTP eventualmente collegati a un backend mock o fake.

Compito dello studente:

1. **Introdurre un backend Rails** che espone le API necessarie.
2. **Collegare Angular al backend reale**, eliminando progressivamente ogni mock.
3. **Estendere** il progetto con almeno una funzionalità avanzata.

4. Requisiti funzionali obbligatori

4.1 Gestione Prodotti

Backend (Rails)

- Endpoints minimi:
    - `GET /products`
        - restituisce elenco prodotti;
        - deve supportare almeno **una** di queste capacità:
            - filtro per categoria/tag (es. `?tag=...`) **oppure**
            - ricerca testuale (es. `?q=...`) **oppure**
            - paginazione (`?page=...&per_page=...`).
    - `GET /products/:id`
        - restituisce il dettaglio del prodotto.

Frontend (Angular)

- `ProductService` deve usare gli endpoint REST reali.
- Componenti di lista e dettaglio **non** devono più contenere dati hardcoded.

- È sufficiente una gestione basilare dei filtri/paginazione coerente con gli endpoint implementati.

4.2 Carrello persistente (non mockato)

Obiettivo: il carrello deve essere **memorizzato sul backend** e ricaricabile.

Backend (Rails)

- Modello dati minimo:
    - `Cart` (associato all'utente, oppure legato a un token se si gestisce carrello guest);
    - `CartItem` con riferimento a `Product`, quantità e prezzo unitario.
- Endpoints minimi:
    - `GET /cart` – restituisce lo stato del carrello corrente;
    - `POST /cart/items` – aggiunge un nuovo prodotto (id + quantità);
    - `PATCH /cart/items/:id` – aggiorna la quantità;
    - `DELETE /cart/items/:id` – rimuove l'articolo dal carrello.

Frontend (Angular)

- `CartService` deve:
    - inizializzare lo stato leggendo `/cart`;
    - aggiornare il backend a ogni modifica (aggiungi/modifica/rimuovi);
    - aggiornare l'UI in maniera coerente.
- Il carrello deve sopravvivere almeno al **reload della pagina** (utente loggato).
- Se lo studente implementa il carrello guest persistito, è considerato un plus.

4.3 Checkout e Ordini

Si parte dalla pagina di checkout già presente (Reactive Forms).

Backend (Rails)

- Modelli minimi:
    - `Order` (dati cliente, totale, data creazione, stato ordine);

    - `OrderItem` (riferimento a prodotto, quantità, prezzo al momento dell'ordine).

- Endpoints minimi:
    - `POST /orders`
        - crea un ordine a partire dal carrello associato all'utente e dai dati del form di checkout;
        - svuota o invalida il carrello dopo la creazione.
    - `GET /orders`
        - restituisce la **lista degli ordini** dell'utente autenticato (anche senza filtri avanzati).
    - `GET /orders/:id`
        - dettaglio di un singolo ordine.

Frontend (Angular)

- `OrderService` deve inviare i dati di checkout a `POST /orders`.
- L'utente deve ricevere un **riscontro chiaro**:
    - stato di loading durante la chiamata;
    - messaggio di conferma in caso di successo;
    - messaggio di errore in caso di fallimento, con eventuale possibilità di retry.
- Una pagina semplice con la lista degli ordini (`GET /orders`) è **obbligatoria** (dettaglio ordini più ricco può rientrare tra le funzionalità avanzate).

4.4 Autenticazione reale (non mockata)

È richiesta **autenticazione reale** lato backend e lato frontend.

Login con credenziali gestito da Rails

- Implementazione con Devise o soluzione custom.
- Endpoints minimi:
    - `POST /auth/login` – riceve credenziali e restituisce un token/jwt o session id;
    - `POST /auth/logout`;
    - `GET /auth/me` – restituisce dati utente corrente.

- Angular:
    - pagina di login con form reale;
    - memorizzazione token e attach via `HttpInterceptor`.

Integrazione nel flusso

- Solo utenti autenticati possono:
    - accedere alla pagina `/checkout`;
    - consultare la lista `/orders`;
    - utilizzare il carrello persistente associato all'utente.
- Angular deve usare almeno una **route guard** per proteggere le rotte riservate.

5. Requisiti tecnici backend (Rails)

- Progetto creato come API (`rails new nome-progetto --api` o configurazione equivalente).
- Database relazionale (preferibilmente PostgreSQL/MySQL o in alternativa Sqlite3).
- Validazioni di base sui modelli (campi obbligatori, formato email, valori numerici positivi, ecc.).
- Gestione errori:
    - uso coerente degli status HTTP (400, 401, 403, 404, 422, 500);
    - risposta JSON che riporti almeno un messaggio di errore significativo.
- **Test minimo richiesto:**
    - almeno 1 test di modello (es. validazione Order o Product);
    - almeno 1 test di request/controller per un endpoint significativo (es. `POST /orders`).

6. Requisiti tecnici frontend (Angular)

Completare quanto visto a lezione con la gestione dello stato HTTP.

- Nessun uso di dati hardcoded per:
    - prodotti;
    - carrello;

    - ordini.

- Servizi dedicati:
    - `ProductService`, `CartService`, `OrderService`, `AuthService` (i nomi possono variare, ma la responsabilità deve essere chiara).
- Uso di **Reactive Forms** per la pagina di checkout (come già visto a lezione).
- Uso di **HttpClient** e **HttpInterceptor** per:
    - token di autenticazione;
    - eventuale gestione centralizzata di errori HTTP.

7. Funzionalità avanzate (obbligatoria almeno 1)

Per calibrare il carico individuale:

- ogni studente deve realizzare **almeno 1 funzionalità avanzata** tra quelle elencate;
- Un'eventuale **seconda funzionalità avanzata** sarà considerata come **bonus** ai fini del voto.

Esempi (non esaustivi):

1. **Area Admin per prodotti**
    - Sezione riservata (protetta) per creare, modificare, cancellare prodotti.
    - Protezione con ruolo `admin` lato backend e guard lato Angular.
2. **Storico ordini avanzato**
    - Pagina “I miei ordini” con filtri per data/stato.
    - Dettaglio ordine con tutte le informazioni (prodotti, quantità, prezzi, indirizzo).
3. **Wishlist / preferiti**
    - Possibilità di aggiungere/rimuovere prodotti da una lista di preferiti.
    - Persistenza lato backend per ogni utente.
4. **Gestione codici sconto / coupon**
    - Modello `Coupon` con regole di validità (date, numero di usi, ecc.).
    - Applicazione del coupon nel checkout con ricalcolo del totale.
5. **Internazionalizzazione**

    - Supporto almeno per Italiano + Inglese.
    - Meccanismo per la scelta della lingua nell'interfaccia.

6. **Test E2E / integrazione**
    - Test unitari:
        - 3-4 Unit test sul backend
        - 2 Unit test su Service Angular
        - 2 Unit test su Component Angular
    - 1–2 scenari end-to-end (es. login → aggiungi al carrello → checkout → ordine).

Lo studente può proporre altre funzionalità avanzate di complessità paragonabile, previa approvazione del docente.

8. Consegna

Ogni studente dovrà consegnare:

1. **Repository Git** (personale) contenente:
    - cartella `backend/` con il progetto Rails;
    - cartella `frontend/` con il progetto Angular.
2. **README principale** con:
    - prerequisiti software (versioni Ruby, Rails, Node, Angular CLI, DB, ecc.);
    - istruzioni passo-passo per:
        - configurare il database (migrazioni, seeding iniziale prodotti);
        - avviare il backend;
        - avviare il frontend in sviluppo e/o servire la build di produzione.
3. Breve documento (può essere un file `ARCHITETTURA.md` o sezione nel README) che descriva:
    - i principali modelli del dominio (Product, Cart, Order, User, ecc.);
    - il flusso login → carrello → checkout → ordine;
    - quali funzionalità avanzate sono state implementate.
4. (Opzionale ma apprezzato) File `docker-compose.yml` per avviare velocemente l'ambiente completo.

9. Prova orale

La valutazione include una **breve discussione orale** del progetto:

- dimostrazione pratica:
    - login reale;
    - visualizzazione prodotti;
    - uso del carrello persistente;
    - checkout con creazione ordine;
    - funzionalità avanzata implementata;
- domande su:
    - scelte progettuali e tecnologiche;
    - modello dati sul backend;
    - integrazione Angular–Rails (servizi, interceptor, guard);
    - gestione errori e sicurezza di base;
- il docente può chiedere allo studente di:
    - spiegare porzioni di codice;
    - apportare una piccola modifica live (es. aggiungere un campo, gestire un nuovo errore).

Criteri di valutazione

Valutazione in trentesimi, indicativamente suddivisa come segue:

1. **Backend Rails**
   API REST, modello dati, validazioni, gestione errori, test minimi → fino a **10 punti**
2. **Frontend Angular**
   integrazione API, gestione carrello/ordini, checkout funzionante → fino a **10 punti**
3. **Funzionalità avanzate**
    - 1 funzionalità avanzata completata: fino a **5 punti**
    - eventuale seconda funzionalità: considerata come **bonus** (margine per 30 e lode)
4. **Qualità complessiva**
   struttura del codice, chiarezza, UX/a11y minima, ordine del repository, documentazione
   → fino a **2 punti**