## Specifiche Funzionali

L'applicazione **Gestione Contatti** è un'app CRUD (Create, Read, Update, Delete) che consente agli utenti di:

1. **Creare** un contatto con nome, cognome ed email.
2. **Leggere** la lista dei contatti salvati.
3. **Eliminare** un contatto dalla lista.
4. Salvare i dati localmente usando **localStorage**.

**Caratteristiche principali:**

- **Interfaccia utente semplice e reattiva**, implementata con HTML/CSS e JavaScript.
- **Persistenza dei dati**: utilizza `localStorage` per mantenere i contatti salvati anche dopo un refresh della pagina.
- **Validazione di base**: il form richiede che tutti i campi siano compilati prima dell'invio.

# Sommario

## Diagramma di Flusso

### Descrizione del flusso di lavoro:

1. L'utente compila il form con i dettagli del contatto.
2. Quando il form viene inviato:
    - I dati vengono validati (campi obbligatori compilati).
    - Il contatto viene aggiunto all'array `contacts`.
    - L'array aggiornato viene salvato in `localStorage`.
3. La lista dei contatti viene aggiornata dinamicamente.
4. Se un utente seleziona "Elimina", il contatto viene rimosso dall'array e `localStorage` viene aggiornato.

Ecco un diagramma di flusso semplificato:

```
 [Start]
   ↓
[Compila Form]
   ↓
[Invia Form]
   ↓                      ↘
[Validazione Campi] → [Errore: Mostra Messaggio]
   ↓
[Aggiungi Contatto all'Array]
   ↓
[Salva in localStorage]
   ↓
[Aggiorna Lista Contatti]
   ↓
[Fine]
```

## Obiettivi

1. **Facilità d'uso:** Fornire un'interfaccia semplice per la gestione dei contatti.
2. **Persistenza dei dati:** Garantire che i contatti rimangano disponibili tra le sessioni dell'utente.
3. **Interattività:** Rinfrescare dinamicamente la lista dei contatti senza ricaricare la pagina.
4. **Manutenzione:** Strutturare il codice in modo modulare per una facile estensione o modifica.

## Analisi Top/Down

### Approccio Top/Down per il design del sistema:

1. **Obiettivo generale:**
    - Creare un'applicazione CRUD per la gestione dei contatti.
2. **Scomposizione in componenti principali:**
    - **Frontend:** Gestisce l'interazione con l'utente.
        - Form HTML per l'inserimento dei dati.
        - Lista dinamica per mostrare i contatti.
    - **Backend simulato:**
        - Array `contacts` per la gestione in memoria.
        - API per leggere, scrivere, ed eliminare dati usando `localStorage`.
3. **Dettagli di implementazione:**
    - **Caricamento iniziale:** Legge i dati da `localStorage` e li visualizza.
    - **Operazioni CRUD:**
        - **Create:** Aggiunge un nuovo contatto all'array.
        - **Read:** Mostra la lista dei contatti.
        - **Delete:** Rimuove un contatto dall'array.
    - **Persistenza:** Sincronizza l'array `contacts` con `localStorage`.

## Simulazione della Memorizzazione su Disco

L'app utilizza **localStorage** per simulare la memorizzazione su disco. Di seguito, l'analisi:

1. **Formato dei dati memorizzati:**
   - I dati vengono salvati come stringa JSON nell'elemento `localStorage` con chiave `contacts`.
   - Esempio:

     ```
     [
       {"name": "Mario", "surname": "Rossi", "email":
     "mario.rossi@example.com"},
       {"name": "Anna", "surname": "Bianchi", "email":
     "anna.bianchi@example.com"}
     ]
     ```

2. **Ciclo di salvataggio e caricamento:**
   - **Salvataggio:** Ogni modifica all'array `contacts` (aggiunta o eliminazione) aggiorna `localStorage`.

     ```
     localStorage.setItem("contacts", JSON.stringify(contacts));
     ```

   - **Caricamento:** All'avvio della pagina, i dati vengono caricati e convertiti in un array.

     ```
     contacts = JSON.parse(localStorage.getItem("contacts"));
     ```

3. **Benefici della simulazione:**
   - Non richiede un server backend.
   - I dati persistono tra le sessioni del browser.
   - Performance elevate per piccole quantità di dati.
4. **Limiti:**
   - **Capacità:** `localStorage` ha un limite di circa 5 MB per dominio.
   - **Sicurezza:** I dati non sono criptati.
   - **Condivisione:** I dati sono specifici per il dispositivo e il browser.

## Conclusione

Questa applicazione CRUD è progettata per essere semplice, efficiente e interattiva, con un'implementazione mirata a introdurre agli utenti i concetti di persistenza locale e manipolazione dinamica del DOM. Può essere ulteriormente estesa con funzionalità come la modifica dei contatti, la sincronizzazione su server remoto o una UI migliorata.

La **programmazione incrementale** è un approccio che consiste nello sviluppare e rilasciare progressivamente le funzionalità di un'applicazione, suddividendo il lavoro in iterazioni successive, ognuna delle quali aggiunge un set di funzionalità completo e testato.

Ecco come applicare questo approccio alla creazione dell'applicazione **Gestione Contatti**:

## Fase 1: Prototipo Base (Interfaccia e Funzionalità Minime)

1. **Obiettivi:**
   - Creare la struttura di base dell'interfaccia utente.
   - Implementare l'aggiunta di un contatto senza memorizzazione.
2. **Cosa implementare:**
   - HTML per il form di input e la lista dei contatti.
   - Un semplice script JavaScript per aggiungere contatti in un array temporaneo.
   - Mostrare i contatti nella lista dinamica sul frontend.
3. **Test:**
   - Verificare che l'utente possa aggiungere contatti e vederli nella lista.
   - Controllare la validazione dei campi (tutti i campi obbligatori compilati).
4. **Output:**
   - Un'applicazione funzionante con un frontend semplice, ma senza persistenza dei dati.

## Fase 2: Persistenza dei Dati (Salvataggio su localStorage)

1. **Obiettivi:**
   - Implementare il salvataggio dei contatti utilizzando `localStorage`.
   - Aggiungere la funzionalità per caricare i dati salvati all'avvio.
2. **Cosa implementare:**
   - Funzione `saveContacts()` per salvare i contatti in `localStorage`.
   - Funzione `loadContacts()` per recuperare i contatti da `localStorage` al caricamento della pagina.
   - Aggiornare le funzioni esistenti per sincronizzare l'array `contacts` con `localStorage`.
3. **Test:**
   - Verificare che i contatti salvati persistano dopo un refresh della pagina.
   - Controllare che i dati vengano salvati e recuperati correttamente.
4. **Output:**
   - L'applicazione mantiene i dati anche dopo la chiusura del browser.

## Fase 3: Funzionalità di Eliminazione

1. **Obiettivi:**
   - Consentire agli utenti di eliminare i contatti dalla lista.
   - Assicurare che i cambiamenti si riflettano in `localStorage`.
2. **Cosa implementare:**
   - Aggiungere un pulsante "Elimina" accanto a ogni contatto nella lista.
   - Funzione `deleteContact(index)` per rimuovere un contatto dall'array.
   - Aggiornare la lista dei contatti e sincronizzare con `localStorage` dopo l'eliminazione.
3. **Test:**
   - Verificare che il contatto selezionato venga rimosso correttamente.

    - Assicurarsi che i dati in `localStorage` vengano aggiornati.
4. **Output:**
    - L'utente può gestire i contatti eliminando quelli non più necessari.

## Fase 4: Miglioramento dell'Interfaccia Utente

1. **Obiettivi:**
    - Rendere l'interfaccia più intuitiva e visivamente accattivante.
    - Fornire feedback visivo per le operazioni (ad esempio, messaggi di conferma o errore).
2. **Cosa implementare:**
    - Aggiornare il design CSS per il form, i pulsanti e la lista dei contatti.
    - Aggiungere messaggi di conferma per operazioni riuscite (es. "Contatto aggiunto con successo").
    - Gestire messaggi di errore per validazione (es. "Email non valida").
3. **Test:**
    - Verificare che l'interfaccia sia chiara e reattiva.
    - Controllare che i messaggi siano visualizzati correttamente.
4. **Output:**
    - Un'applicazione con un'interfaccia moderna e più user-friendly.

## Fase 5: Modularità e Pulizia del Codice

1. **Obiettivi:**
    - Rifattorizzare il codice per migliorarne la leggibilità e la manutenibilità.
    - Separare la logica di business dal rendering del frontend.
2. **Cosa implementare:**
    - Creare funzioni più modulari (es. separare il rendering dalla logica CRUD).
    - Isolare il codice legato a `localStorage` in funzioni dedicate.
    - Documentare il codice con commenti e migliorare la struttura complessiva.
3. **Test:**
    - Verificare che l'applicazione funzioni ancora correttamente dopo la rifattorizzazione.
    - Assicurarsi che il codice sia leggibile e facile da comprendere.
4. **Output:**
    - Un codice ben strutturato e pronto per l'estensione.

## Fase 6: Test Completo e Debugging

1. **Obiettivi:**
    - Testare l'applicazione in scenari reali e risolvere eventuali bug.
    - Garantire che tutte le funzionalità funzionino correttamente.
2. **Cosa implementare:**
    - Testare l'aggiunta, l'eliminazione e la persistenza dei contatti su diversi browser.
    - Gestire eventuali errori edge-case, come `localStorage` pieno o dati corrotti.
3. **Test:**
    - Simulare errori, ad esempio cancellando i dati di `localStorage` manualmente.
    - Verificare la compatibilità su diversi dispositivi e browser.
4. **Output:**
    - Un'applicazione stabile e senza bug.

## Fase 7: Espansioni Future (Optional)

1. **Obiettivi:**
    - Implementare nuove funzionalità basate sui feedback degli utenti.
2. **Cosa implementare (idee):**
    - Funzionalità di modifica dei contatti.
    - Ricerca e filtro dei contatti.
    - Sincronizzazione con un database remoto (es. tramite API REST).
3. **Test:**
    - Verificare l'integrazione di nuove funzionalità senza interrompere quelle esistenti.
4. **Output:**
    - Un'app completa e avanzata con nuove funzionalità.

## Conclusione

Seguendo la programmazione incrementale, si costruisce progressivamente un'applicazione stabile, modulare e facile da manutenere, iniziando dalle basi fino a funzionalità più complesse, con continui cicli di test e miglioramento.