Esercizi pratici: fetch, Promise, CRUD

Niccolò Maltoni
niccolo.maltoni@diennea.com

Obiettivo della lezione

In questa lezione riprendiamo il ricettario costruito nelle lezioni 5 e 6 e lo adattiamo per comunicare con un'API remota.

L'interfaccia rimane quasi la stessa - stesso HTML, stesso CSS, stessa logica di navigazione - ma al posto di `localStorage` useremo `fetch` con `async/await` per leggere, creare, aggiornare ed eliminare ricette su un server.

Le uniche aggiunte strutturali sono il campo `id`, restituito dall'API, e il pulsante per eliminare la ricetta corrente.

Da localStorage a fetch

Nella lezione 6 usavamo `salvaRicettario()` e `caricaRicettario()` per persistere i dati nel browser.

Oggi questi due meccanismi vengono sostituiti da quattro funzioni asincrone, una per ogni operazione CRUD:

Operazione	**Prima (lez. 6)**	**Ora (lez. 9)**
Lettura	`caricaRicettario()` da `localStorage`	`fetch` GET
Creazione	push locale + `salvaRicettario()`	`fetch` POST
Modifica	aggiornamento locale + `salvaRicettario()`	`fetch` PUT
Eliminazione	non prevista	`fetch` DELETE

L'API ci restituisce anche un campo `id` per ogni ricetta: aggiungiamolo subito al costruttore di `Recipe`, perché ci servirà in PUT e DELETE.

L’API mock

I dati delle ricette vivono in un file `db.json` ospitato su GitHub. Usiamo **my-json-server** di typicode per esporlo come API REST:

```
https://my-json-server.typicode.com/NiccoMlt/demo-ricettario-api/recipes
```

La struttura di ogni ricetta è la stessa che conosciamo, con l’aggiunta di `id`:

```
{
  "id": 1,
    "name": "Spaghetti alla Carbonara",
    "image": "https://raw.githubusercontent.com/.../carbonara.png",
    "description": "Un classico della cucina romana...",
    "ingredients": { "g di spaghetti": 80, "Pepe nero": null },
    "preparation": ["Mettere a bollire l'acqua...", "..."]
}
```

Esercizio guidato: CRUD asincrono

Implementeremo il ricettario in quattro step incrementali, uno per ogni operazione.

Prima di passare allo step successivo, proviamo sempre a testare quello che abbiamo scritto.

Punto di partenza

Possiamo ripartire dall'esercizio completato della lezione 6 e trasformarlo passo passo.

Per rendere il materiale più coerente, in questo repository lo starter è già stato riscritto con identificatori e struttura dati in inglese, mantenendo i testi della pagina in italiano.

Il nostro compito resta sostituire la persistenza locale con le quattro operazioni di rete e completare il pulsante di eliminazione.

Step 1: GET — carica le ricette

Al caricamento della pagina dobbiamo recuperare le ricette dall'API e popolare il ricettario.

Usiamo `await` per aspettare la risposta prima di procedere, e controlliamo `response.ok` perché `fetch` non lancia errori automaticamente per i codici HTTP 4xx/5xx.

```
async function caricaRicette() {
    const response = await fetch(API_URL);
    if (!response.ok) throw new Error(`HTTP ${response.status}`);
    const dati = await response.json();
    recipeBook.recipes = dati.map(r =>
        new Recipe(r.id, r.name, r.image, r.description, r.ingredients, r.preparation)
    );
    recipeBook.currentIndex = 0;
    showRecipe();
}

loadRecipes();
```

Step 2: POST — crea una ricetta

Quando l'utente invia il form di creazione, mandiamo i dati all'API con POST.

L'API risponde con la ricetta appena creata, incluso l'`id` assegnato dal server: lo conserviamo subito nell'oggetto locale.

```
async function creaRicetta(dati) {
    const response = await fetch(API_URL, {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(dati)
    });
    if (!response.ok) throw new Error(`HTTP ${response.status}`);
    const creata = await response.json();
    recipeBook.add(new Recipe(
        creata.id, creata.name, creata.image,
        creata.description, creata.ingredients, creata.preparation
    ));
    recipeBook.currentIndex = recipeBook.total() - 1;
    showRecipe();
}
```

Step 3: PUT — aggiorna una ricetta

Ogni volta che aggiungiamo un ingrediente o un passo, aggiorniamo la ricetta sul server con PUT.

Modifichiamo prima l'oggetto locale, poi inviamo l'intera ricetta aggiornata:

```
async function updateRecipe(id, recipe) {
    const response = await fetch(`${API_URL}/${id}`, {
        method: 'PUT',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(recipe)
    });
    if (!response.ok) throw new Error(`HTTP ${response.status}`);
    showRecipe();
}
```

Nel form ingrediente, per esempio:

```
recipeBook.current().ingredients[name] = quantity ? Number(quantity) : null;
await updateRecipe(recipeBook.current().id, recipeBook.current());
```

Step 4: DELETE — elimina una ricetta

Per eliminare una ricetta usiamo DELETE con l'`id` della ricetta corrente.

Dopo la conferma dal server, la rimuoviamo anche dall'array locale e aggiorniamo la navigazione.

```
async function deleteRecipe(id) {
    const response = await fetch(`${API_URL}/${id}`, { method: 'DELETE' });
    if (!response.ok) throw new Error(`HTTP ${response.status}`);
    const index = recipeBook.currentIndex;
    recipeBook.recipes.splice(index, 1);
    recipeBook.currentIndex = Math.min(index, recipeBook.total() - 1);
    showRecipe();
}
```

Consiglio

`Math.min(indice, totale - 1)` evita che l'indice superi i limiti dell'array dopo l'eliminazione: se cancelliamo l'ultima ricetta, torniamo automaticamente alla penultima.

Endpoint di riferimento

Metodo	Endpoint	Azione
GET	`/recipes`	tutte le ricette
GET	`/recipes/{id}`	una ricetta
POST	`/recipes`	aggiungi ricetta
PUT	`/recipes/{id}`	aggiorna ricetta
DELETE	`/recipes/{id}`	elimina ricetta

> **Nota**
>
> L'API mock simula tutte le operazioni CRUD e restituisce risposte coerenti, ma non salva davvero le modifiche in modo permanente.