Alma Mater Studiorum · Università di Bologna

Dipartimento di Informatica — Scienza e Ingegneria
Corso di Laurea Triennale in Informatica

ASYNC/AWAIT VS GENERATOR/YIELD

Modelli di controllo del flusso asincrono in JavaScript

Relatore:
Chiar.mo Prof.
Fabio Vitali

Presentata da:
Luca Paparo

Sessione Marzo 2026
Anno Accademico 2024/2025

Abstract

L'evoluzione dei modelli asincroni in **JavaScript** ha seguito un percorso di crescente astrazione per arrivare alle attuali **Promise** e al paradigma `async/await`, ma non ha ancora risolto alcuni limiti strutturali legati al controllo del flusso asincrono, sia da un punto di vista di gestione che di leggibilità.
Questi limiti emergono con particolare evidenza nelle applicazioni moderne, dove la prevedibilità data dalla leggibilità stessa del comportamento asincrono è fondamentale.
Mostreremo in questa tesi che il linguaggio possiede già nativamente, anche se solo parzialmente, gli strumenti per superare tali limiti: le funzioni generatore, tramite il paradigma `generator/yield`.
In questo contesto si colloca `Effection`, un runtime che sfrutta i generatori per introdurre in **JavaScript** la concorrenza strutturata, la quale rende il comportamento asincrono più prevedibile e coerente con l'intenzione dello sviluppatore.
Infine, evidenzio come né `Effection` né il paradigma `generator/yield` rappresentino un punto di arrivo definitivo, ma una direzione: un nuovo paradigma per l'asincronicità che combini la granularità di `generator/yield` con la leggibitilà e concorrenza strutturale di `Effection`.

Indice

Capitolo 1

INTRODUZIONE

L'evoluzione dei modelli asincroni in **JavaScript** ha seguito un percorso lungo e articolato, caratterizzato da soluzioni che hanno progressivamente migliorato la sua espressività.
Nonostante i progressi introdotti da **Promise** e `async/await`, il linguaggio presenta ancora dei limiti strutturali nella gestione del flusso asincrono, soprattutto quando si considerano applicazioni moderne e concorrenti.
I problemi che affronteremo riguardano principalmente le difficoltà di modellare operazioni asincrone in modo leggibile, prevedibile e naturale, in modo tale che si capiscano le reali intenzioni del developer già dal livello sintattico e che il codice scritto abbia una leggibitilà simile a quella delle operazioni sincrone.
In secondo luogo osserveremo la difficoltà di implementare operazioni asincrone componibili e strutturate, come per esempio in delle pipeline, utilizzando solo **Promise** con `async/await`.
Vedremo delle possibili soluzioni, nello specifico il paradigma `generator/yield` e la libreria `Effection`.
Da un lato, i generatori rappresentano il paradigma più granulare disponibile in `JavaScript` per modellare il codice asincrono, dall'altro, `Effection` dimostra come sia possibile costruire sopra i generatori un runtime che offra la concorrenza strutturata, rendendo il comportamento asincrono molto più prevedibile, simile al codice sincrono.

Capitolo 2

Paradigma async/await

2.1 Introduzione

Il modello `async/await` rappresenta ad oggi la forma più diffusa per gestire operazioni asincrone in **JavaScript**.
Introdotto con `ECMAScript 2017`[5], esso si basa sulle **Promise**, ma fornisce una sintassi più lineare e vicina allo stile del codice sincrono.
L'obiettivo è ridurre la complessità della gestione del flusso di controllo, semplificare la gestione degli errori e rendere più chiara la composizione di operazioni.
Per comprendere appieno il funzionamento di `async/await`, è necessario ripercorrere brevemente l'evoluzione dei meccanismi asincroni in **JavaScript**: dalle **callback**, alle **Promise**, fino all'utilizzo di **await**.

2.2 Callback

Le **callback** sono state il primo meccanismo asincrono disponibile in **JavaScript**. Una **callback** è una funzione passata come argomento ad un'altra, in modo da poter essere eseguita ad un momento opportuno.
Un esempio di **callback**:

```
function readSensor(callback) {
  setTimeout(() => {
    const value = Math.random() * 10;
    callback(null, value);
  }, 500);
}

readSensor((err, value) => {
  if (err) {
    console.error("Errore:", err);
    return;
  }
  console.log("Valore:", value);
});
```

In questo esempio:

1. **setTimeout** ritarda l'esecuzione del blocco di codice interno di 500 millisecondi
2. Nel blocco di codice interno, il parametro **callback** preso in input viene chiamato con parametri **null** (eventuale errore) e **value**
3. **readSensor** viene quindi chiamata passandogli come parametro una funzione, la quale verrà eseguita al termine dei 500 millisecondi da **callback(null, value)**

2.3 Promise

Le `Promise` rappresentano il fondamento dell'intero modello asincrono moderno di **JavaScript**.
Essa è un oggetto che rappresenta il risultato di un'operazione asincrona.
Può trovarsi in uno dei seguenti stati:

- **pending**: l'operazione è in corso
- **fulfilled**: l'operazione è completata con successo
- **rejected**: l'operazione è fallita

Un esempio di funzione che ritorna una `Promise`:

```
function readSensor() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const value = Math.random() * 10;
      if (value < 1) reject("Errore sensore");
      else resolve(value);
    }, 500);
  });
}
```

In questo esempio:

1. **readSensor** ritorna semplicemente una **Promise**
2. La **Promise** ha a disposizione due **callback**, **resolve** e **reject**.
    - **resolve(value)**: indica il successo, ritornando il valore `value`
    - **reject(error)**: indica il fallimento, lanciando l'errore `error`

Quando una **Promise** viene completata (detto **settled**, ossia quando termina con successo o solleva un'eccezione), i **callback** registrati tramite `.then()` o `.catch()` verranno eseguiti:

```
function readSensor() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const value = Math.random() * 10;
      if (value < 1) reject("Errore sensore");
      else resolve(value);
    }, 500);
  });
}

readSensor()
  .then(value => {
    console.log("Valore:", value);
    return value * 2;
  })
  .catch(err => {
    console.error("Errore:", err);
  });
```

Le **Promise** non sono solo un modo per gestire l'asincronia, ma introducono un livello di controllo del flusso più fine rispetto alle callback tradizionali.

2.4 Async/Await

Il modello `async/await` rappresenta ad oggi la forma più comune per gestire l'asincronia in **JavaScript**.
Pur essendo ai tempi percepito come un meccanismo nuovo, esso non introduce un paradigma asincrono autonomo: si tratta di una sintassi più comoda costruita sopra le **Promise**.
Per questo motivo, comprendere `async/await` significa comprendere come **JavaScript** gestisce la sospensione, la ripresa dell'esecuzione e la propagazione degli errori.

2.4.1 Le funzioni asincrone

Una funzione dichiarata con la parola chiave **async** resituirà sempre una **Promise**.

```
async function readSensor() {
  return 42;
}

readSensor().then(v => console.log(v)); // stampa 42
```

Utilizzando solo **Promise**, la funzione `readSensor` è equivalente a:

```
function readSensor() {
  return Promise.resolve(42);
}
```

Le **callback resolve** e **reject** diventano implicite.

2.4.2 L'operatore await

La parola chiave **await** fa in modo che non venga ritornato il controllo alla funzione chiamante fino a che l'esecuzione della **Promise** non è completata.

```
async function readSensor() {
    return 42;
}
const value = await readSensor();
console.log(value); // stampa 42
```

2.5 Esempio: misura di sensori

Immaginiamo ci serva ricevere tre misure da tre sensori diversi:

- temperatura
- umidità
- pressione

Possiamo farlo in modi diversi.

2.5.1 Sequenziale

```
async function readTemperature() {
    return 41;
}

async function readHumidity() {
    return 42;
}

async function readPressure() {
    return 43;
}

async function readAllSensors() {
  const t = await readTemperature();
  const h = await readHumidity();
  const p = await readPressure();
  return { t, h, p };
}
```

In questo caso, ogni **await** aspetta il precedente, le **Promise** vengono eseguite una alla volta.

2.5.2 Concorrentemente

```
const [t, h, p] = await Promise.all([
  readTemperature(),
  readHumidity(),
  readPressure()
]);
```

In questo caso utilizziamo invece il metodo statico **Promise.all**, il quale prende un iterabile di **Promise** e ritorna una singola **Promise** che viene risolta se e e solo se tutte le **Promise** passate nell'iterabile risolvono.

2.6 Gestione del flusso

Come abbiamo visto, il modello `async/await` semplifica notevolmente la scrittura di codice asincrono rispetto alle semplici **callback**, ma questa semplicità conseguono una serie di compromessi strutturali.
Questi limiti emergono soprattutto quando si costruiscono pipeline complesse, per esempio quando si desidera reagire dinamicamente a modifiche del flusso di controllo.

Analizziamo quindi come funziona in questo paradigma la gestione degli errori e proviamo a vedere cosa succede alla gestione del flusso provando a creare un semplice **Retry system**.

2.6.1 Gestione degli errori

A prima vista la gestione degli errori con `async/await` sembra banale: basta usare i blocchi di codice `try...catch` facendo chiamate sequenziali e il codice appare lineare.
In realtà, proprio il modo in cui gli errori interrompono il flusso rende il modello poco flessibile in pipeline complesse.

```
async function pipeline() {
  try {
    const t = await readTemperature();
    const h = await readHumidity();
    const p = await readPressure();
    return { t, h, p };
  } catch (err) {
    console.error("Errore nella pipeline:", err);
  }
}
```

- Il codice dentro il blocco **try** viene eseguito
- In caso qualcosa all'interno del blocco **try** generi un'eccezione, viene eseguito il blocco **catch**

In questo caso, assumendo **readPressure** lanci un eccezione:

1. interrompe immediatamente la funzione
2. salta tutte le istruzioni successive
3. viene catturata dal **catch**, che vederà l'errore lanciato dalla funzione asincrona che ha fallito

Con un solo blocco `try...catch`, non è possibile, per esempio, assumendo sempre che `readPressure` generi un'eccezione:

- riprendere l'esecuzione da **readPressure()**
- iniettare un valore alternativo e continuare
- ripetere solo lo step fallito senza rieseguire tutto.

Proviamo ora con piú `try...catch` annidati:

```
async function pipeline() {
  let t;
  try {
    t = await readTemperature();
  } catch (err) {
    console.error("Errore temperatura:", err);
  }

  let h;
  try {
    h = await readHumidity();
  } catch (err) {
    console.error("Errore umidita:", err);
  }

  let p;
  try {
    p = await readPressure();
  } catch (err) {
    console.error("Errore pressione:", err);
    throw err; // qui invece fallisco
  }
  return { t, h, p };
}
```

Qui si vede chiaramente il limite del paradigma: quando si vuole gestire un caso complesso, **la logica applicativa si mescola con la logica di controllo**:

- Ogni politica riguardo gli errori è scritta a mano in ogni punto dove la funzione può fallire
- Non esiste un luogo centralizzato dove gestire tutti gli errori generati dalla funzione, ma viene spalmato su ogni chiamata di funzione

2.6.2 Retry system

Il retry è uno degli aspetti in cui il modello `async/await` mostra più chiaramente i suoi limiti.
In un sistema reale, come una pipeline di acquisizione dati, è comune che alcune operazioni falliscano in modo intermittente, come per esempio a causa di un sensore guasto.
Un modello asincrono robusto dovrebbe permettere di:

- Ripetere solo lo step fallito o permettere di ignorarne il fallimento se possibile
- Applicare politiche di retry **facilmente configurabili**
- **Separare la logica applicativa dalla logica di controllo**
- Mantenere il flusso **leggibile** e **componibile**

Il modello `async/await` non offre alcun meccanismo nativo per il retry, va implementato manualmente.
Per ripetere una singola operazione, per esempio, è necessario scrivere un wrapper esplicito e poi ricordarsi di usarlo:

```
async function retry(fn, attempts = 3) {
  for (let i = 0; i < attempts; i++) {
    try {
      return await fn();
    } catch (err) {
      if (i === attempts - 1) throw err;
    }
  }
}
const temperature = await retry(readTemperature);
```

Questa funzione **retry**, prende in input una funzione ***fn*** e riprova fino a ***attempts*** volte (default 3) la funzione asincrona ***fn*** passata in input.
Problematiche:

- Il retry non è trasparente, ogni chiamata deve essere modificata manualmente passandola alla funzione **retry**
- La logica applicativa continua ad essere contaminata dalla logica di controllo

Per evitare di dover modificare a mano ogni chiamata asincrona presente nella pipeline, proviamo invece ad implementare il meccanismo di retry sulla pipeline stessa:

```
async function runPipeline() {
  for (let i = 0; i < 3; i++) {
    try {
      return await pipeline();
    } catch (err) {
      console.log("Tentativo fallito:", i + 1);
    }
  }
  throw new Error("Pipeline fallita");
}
```

Problematiche:

- Ripeti anche gli step che erano riusciti
- Non puoi ripetere solo lo step fallito
- Non puoi riutilizzare la pipeline in contesti diversi (e.g. ho fissato il comportamento in caso di errore, ma magari vorrei un comportamento diverso in base al contesto)
- Se aggiungi uno step, devi aggiornare tutto (e.g. aggiungo uno step a cui magari non serve retry o ha richieste diverse dagli altri)

2.7 Possibili strade risolutive

Uno dei problemi principali di ciò che abbiamo visto è l'utilizzo diretto delle **Promise**, soprattutto tramite **await**: nel momento in cui ci fermiamo ad aspettare il codice asincrono in questo modo, diventa impossibile gestire in maniera leggibile e componibile il risultato, soprattutto in caso di errore (un altro motivo è il cosiddetto **Await Event Horizon** che vedremo nel capitolo successivo 3).
Presentiamo quindi due modi per la risoluzione del problema: una è la libreria `Effection`[7;6], la quale usa la concorrenza strutturata oltre a non permettere l'utilizzo diretto di **Promise**, ma solo di iteratori e funzioni generatore.
L'altro modo è utilizzare invece le funzioni generatore nativamente senza alcuna libreria[8;9] ma solo con ciò che è offerto , utilizzando comunque le **Promise** ma nella maniera più indiretta possibile.

Capitolo 3

Effection

3.1 Introduzione

`Effection` è una libreria **JavaScript** che implementa un modello di concorrenza strutturata basato sulle funzioni generatore.
È sviluppata da **Frontside**[1] e nasce con l'obiettivo di fornire un modello di gestione della asincronicità coerente e prevedibile e in cui il flusso di controllo rispetta l'ordine lessicale in cui è definito.
`Effection`, in quanto libreria, non introduce nuove primitive del linguaggio ma si basa esclusivamente sulle funzioni generatore.

3.2 La concorrenza strutturata

La concorrenza strutturata enunciata da `Effection` garantisce due cose:

Nessuna operazione dura più a lungo dell'operazione padre

In **JavaScript** ogni operazione asincrona viene eseguita fintanto che gli è necessario. Questo significa che, se un'operazione "padre" asincrona esegue al suo interno un'altra operazione asincrona (operazione "figlia"), anche se la funzione padre viene interrotta, l'esecuzione della "figlia" continua fino al completamento.
In `Effection`, **ogni operazione asincrona viene eseguita fintanto che è necessaria**, pertanto, se un'operazione "padre" è conclusa, sono concluse anche tutte le operazioni "figlie".

[1] `https://frontside.com`

Ogni operazione si conclude completamente

Una limitazione del **Javascript** del paradigma `async/await` è il cosiddetto **Await Event Horizon**[2].
Ogni volta che si usa **await** per attendere il risultato di una **Promise**, il controllo non ritorna alla funzione chiamante fino al suo completamento.
`Effection` garantisce che le funzioni ritornino un valore o lancino un errore, così come succede per le operazioni sincrone.

Stele di rosetta per Effection

`Effection` dichiara una vera e propria stele di Rosetta per fare chiarezza sulle somiglianze con il paradigma `async/await` per come si utilizza lessicalmente.
Vediamo quindi alcune delle corrispondenze, con `async/await` a sinistra e `Effection` a destra:

- **await** è equivalente a **yield**
- **async function** è equivalente a **function***
- **Promise** è equivalente a **Operation**

Scope

Ogni operazione in `Effection` si svolge nel contesto di **scope** che impone un limite alla sua durata.
Lo script sottostante usa l'operazione **spawn()** (la quale permette di lanciare un'altra operazione in maniera concorrente e figlia dell'operazione attuale) per eseguire contemporaneamente un'operazione che conta fino a dieci, producendo un numero ogni secondo, per poi rimanere in sospeso per cinque secondi prima di ritornare.

```
1  import { main, sleep, spawn } from "effection";
2
3  await main(function* () { // scope padre
4    yield* spawn(function* () { // scope figlio
5      for (let i = 1; i < 10; i++) {
6        console.log(i);
7        yield* sleep(1000);
8      }
9    });
10   yield* sleep(5000);
11 });
```

[2] https://frontside.com/blog/2023-12-11-await-event-horizon/

Questo script genera solo cinque numeri, non dieci.
Questo perché l'operazione padre si completa dopo soli cinque secondi e, di conseguenza, il suo **scope**, e ogni **scope** contenuto in esso, viene interrotto.
Questo perché **nessuna operazione può vivere di più del suo scope**.
Da questa regola ne consegue che si può creare ogni tipo di processo dinamico e concorrente, ma nel momento in cui esce dallo **scope** e non è più necessario, verrà immediatamente interrotto.

3.3 Gestione del flusso

Prendiamo l'esempio di prima, traducendolo in **JavaScript**

```
function sleep(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function childTask() {
  for (let i = 1; i <= 10; i++) {
    console.log(i);
    await sleep(1000);
  }
}

async function parentTask() {
  childTask();
  await sleep(5000);
}

parentTask();
```

A differenza della versione con `Effection`, eseguendo questo script, vedremo chiaramente che la funzione **childTask()** viene portata a termine e non viene interrotta, stampando quindi dieci numeri.
Questo non è corretto siccome non rispetta le regole dello scope lessicale.
In **JavaScript** non posso fare riferimento a una variabile locale al di fuori del suo scope:

```
{
  let risposta = 42;
}
console.log(risposta) // ReferenceError: risposta is not defined
```

Questo generalizza il comportamento già presente dello scope lessicale in **JavaScript** estendendolo alle funzioni, rendendo l'esecuzione del codice più prevedibile grazie anche alla sola lettura.

Vediamo come viene tradotta la pipeline vista nel capitolo 2.6 in `Effection`:

```
import { until, main } from 'effection';

function* pipeline() {
  const t = yield* until(readTemperature());
  const h = yield* until(readHumidity());
  const p = yield* until(readPressure());
  return { t, h, p };
}
await main(function*(){
  yield* pipeline()
})
```

La chiamata a **main** è il punto d'ingresso per il runtime offerto da `Effection`.
Tutti i programmi scritti con `Effection` iniziano chiamando il metodo **main** una volta e tutto ciò che il programma deve fare va gestito al suo interno.
La funzione **until** permette di convertire una **Promise** in un **Operation** equivalente, che il tipo centrale su cui lavora `Effection`.
Operation identifica un'operazione astratta e, come per `generator/yield`, non fa niente autonomamente ma solo quando viene eseguita tramite **yield***.

Uno dei principali vantaggi è della pipeline scritta in `Effection` è che, non ritornando **Promise**, **non esiste il problema del "bubbling up"** (e.g. ritornando una **Promise** farebbe si che la funzione chiamante debba gestirla ritornando una **Promise** a sua volta)
Possiamo anche scriverla in maniera concorrente e non sequenziale usando l'operatore **all**, che permette di aspettare il completamento di tutte le operazioni:

```
import { until, main } from 'effection';

function* pipeline() {
    const [t, h, p] = yield* all([
      until(readTemperature()),
      until(readHumidity()),
      until(readPressure()),
    ]);
    return { t, h, p };
}
await main(function*(){
  yield* pipeline()
})
```

In questo caso abbiamo una versione concorrente che però ha le garazie offerte dalla concorrenza strutturale, pertanto nel caso fallisca anche una sola di quelle operazioni, verrà terminato lo **scope** padre e pertanto anche le altre operazioni figlie.

3.4 Gestione degli errori

Uno dei motivi principali per cui `Effection` è stata scritta è proprio per migliorare la gestione degli errori rispetto al come funziona **JavaScript** nativamente[3].
Come abbiamo visto precedentemente, in `Effection`, al fallimento di un operazione figlia, fallisce l'operazione padre.
Questo segue intuitivamente il funzionamento delle funzioni sincrone in cui l'errore si propaga lungo lo stack.
La miglioria più importante rispetto ad **async/await** è che possiamo usare semplicemente blocchi `try...catch` invece di **Promise** e **callback**, e di non dover creare costrutti ad hoc per ottenere questo risultato.
Una nota importante su cui si sofferma anche la guida offerta da `Effection` è la seguente:

```
import { main, suspend } from 'effection';

function* tickingBomb() {
  yield* sleep(1000);
  throw new Error('boom');
}

await main(function*() {
  yield* spawn(tickingBomb);
  try {
    yield* suspend();
  } catch(err) {
    console.log("it blew up:", err.message);
  }
});
```

La guida fa notare che in questo caso non entriamo nel blocco **catch** ma che il processo dentro il **main** fallisce a causa del **throw new Error('boom')**.
Questo perché **spawn** lancia un operazione concorrente figlia a cui non facciamo **yield** direttamente.
Diventa quindi più facile capire tramite la sola lettura del codice quali errori possono essere catturati dal blocco **catch**.
Se però vogliamo catturare gli errori di un'operazione concorrente, dobbiamo farlo esplicitamente. Per questo esiste la funzione **scoped**:

[3] `https://frontside.com/effection/guides/v4/errors/`

```
import { main, scoped, spawn, suspend } from 'effection';

function* tickingBomb() {
  yield* sleep(1000);
  throw new Error("boom");
}

await main(function* () {
  try {
    yield* scoped(function* () {
      yield* spawn(tickingBomb);
      yield* suspend();
    });
  } catch (err) {
    console.log("it blew up:", err.message);
  }
});
```

In questo modo al fallimento di **tickingBomb**, l'errore sarà propagato alla funzione padre.

Capitolo 4

Paradigma generator/yield

4.1 Introduzione

Le funzioni generatore rappresentano uno dei meccanismi più flessibili introdotti in **JavaScript** con `ECMAScript 2015`[4].
Questa è proprio la base su cui `Effection` ha costruito le sue funzionalità.
A differenza delle funzioni tradizionali, che eseguono il proprio corpo dall'inizio alla fine senza interruzioni, una funzione generatore può sospendere la propria esecuzione e riprenderla in un secondo momento.
Questa caratteristica rende i generatori uno strumento estremamente potente per modellare flussi di controllo complessi e orchestrare operazioni asincrone in modo più granulare rispetto a `async/await`.

4.2 Iteratori

Gli iteratori rappresentano uno dei concetti fondamentali del linguaggio **JavaScript** e, più in generale, della programmazione moderna.
Un **iteratore** è un oggetto che definisce una sequenza e possibilmente ritorna un valore al termine di essa.[1]
Esso implementa un metodo chiamato **next()**, il quale ritorna un oggetto avente due proprietà:

- **value**: il valore corrente della sequenza
- **done**: un booleano che indica se la sequenza è terminata

Questo modello è alla base di costrutti come `for...of`, degli oggetti iterabili (`Array`, `Map`, `Set`) e, soprattutto, delle funzioni generatore.

Questo è un esempio minimale per comprendere come funziona un iteratore:

```
const iterator = {
  current: 0,
  next() {
    this.current++;
    return {
      value: this.current > 3 ? undefined : this.current,
      done: this.current > 3
    };
  }
};

iterator.next(); // { value: 1, done: false }
iterator.next(); // { value: 2, done: false }
iterator.next(); // { value: 3, done: false }
iterator.next(); // { value: 4, done: true }
```

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_generators

4.3 Funzioni generatore

In **JavaScript**, una funzione generatore è una funzione che:

- Si dichiara con la parola chiave **function***
- Restituisce un **generatore**
- Può sospendere la sua esecuzione tramite la parola chiave **yield**
- L'esecuzione della funzione può essere ripresa chiamando **.next()** sul **generatore** ritornato

Esempio di funzione generatore:

```
function* numbers() {
  yield 1;
  yield 2;
  yield 3;
}
```

L'esecuzione non avviene autonomamente ma si ha il controllo su di essa chiamando **.next()**, iterando su tutti i valori restituiti dalla funzione tramite **yield**

```
const it = numbers();
it.next(); // { value: 1, done: false }
it.next(); // { value: 2, done: false }
it.next(); // { value: 3, done: false }
it.next(); // { value: undefined, done: true }
```

4.4 L'istruzione yield

L'istruzione **yield**:

1. Sospende l'esecuzione della funzione
2. Restituisce un valore al chiamante
3. Attende che il chiamante richiami **.next()**, (passando un eventuale valore se necessario)
4. Riprende l'esecuzione dal punto esatto in cui era stata sospesa

yield è quindi la funzionalità che ci permette di effettuare il controllo del flusso della funzione interna direttamente dall'utilizzatore dell'iteratore e non da dentro la pipeline stessa.
Scriviamo **yield*** quando il valore da ritornare è generato da un'altra funzione generatore.

4.5 Gestione del flusso

La differenza fondamentale tra `async/await` e `generator/yield` non è solo sintattica, ma concettuale.
Come abbiamo visto nel capitolo 2.6, con `async/await` la gestione del flusso di controllo è interna alla funzione, mentre con `generator/yield` possiamo delegare la gestione del flusso di controllo all'utilizzatore del generatore (che chiameremo, e abbiamo già chiamato, **scheduler**, siccome è effettivamente chi usa il generatore che *schedula* i risultati tornati dall'iteratore).
Questa distinzione cambia radicalmente ciò che è possibile fare.
Per mostrare questa differenza, analizziamo lo stesso esempio implementato precedentemente con `async/await`, ora riscritto con generatori, e vediamo come cambia la gestione del flusso di controllo.

Un generatore può ritornare delle `Promise`:

```
function* pipeline() {
  const result = yield Promise.resolve(42);
  return result;
}
```

Pertanto, la seguente pipeline scritta in `async/await`

```
async function pipeline() {
  const t = await readTemperature();
  const h = await readHumidity();
  const p = await readPressure();
  return { t, h, p };
}
```

diventa, utilizzando `generator/yield`

```
function* pipeline() {
  const t = yield readTemperature();
  const h = yield readHumidity();
  const p = yield readPressure();
  return { t, h, p };
}
```

A differenza della versione `async/await`, la versione `generator/yield` **non esegue niente in maniera autonoma**, per cui andremo sempre a **dividere la logica applicativa dalla gestione del flusso**.
L'unica limitazione che abbiamo in questo caso è che, siccome stiamo usando **Promise**, esse vengono comunque eseguite nel momento in cui facciamo **yield**.

Vediamo uno **scheduler** minimale che esegue una pipeline basata su generatori:

```
function* pipeline() {
  const t = yield readTemperature();
  const h = yield readHumidity();
  const p = yield readPressure();
  return { t, h, p };
}

function run(gen) {
  const it = gen();

  function step(lastValue) {
    const { value, done } = it.next(lastValue);

    if (done) return value;

    return value.then(
      res => step(res),
      err => it.throw(err)
    );
  }

  return step();
}

run(pipeline);
```

Questa funzione **run**:

1. Istanzia il generatore **gen** passato in input
2. Chiama ricorsivamente la funzione **step** su ogni elemento generato dall'iteratore, scorrendo quindi tutti i risultati generati da **yield**
3. **.next(lastValue)** inietta nella pipeline il valore restituiti dall'operazione yieldata e ne riprende l'esecuzione. Altrimenti, in caso di errore, il generatore solleva un'eccezione tramite **.throw()**
4. Quando la pipeline farà **return**, **done** sarà true e la funzione **run** ritornerà il risultato finale.

4.5.1 Gestione degli errori

La differenza fondamentale rispetto ad `async/await` che **l'errore non interrompe obbligatoriamente la pipeline**, poiché il flusso di controllo non è interno alla funzione, ma è nello **scheduler**.

Riprendiamo l'esempio del capitolo precedente:

```
function* pipeline() {
  const t = yield readTemperature();
  const h = yield readHumidity();
  const p = yield readPressure();
  return { t, h, p };
}

function run(gen) {
  const it = gen();

  function step(lastValue) {
    const { value, done } = it.next(lastValue);

    if (done) return value;

    return Promise.resolve(value).then(
      res => step(res),
      err => it.throw(err)
    );
  }

  return step();
}

run(pipeline);
```

Tramite la funzione **run**, abbiamo pieno controllo sul modo in cui vogliamo gestire la pipeline.
In questo caso, **it.throw(err)** rilancia l'errore dentro il generatore, nello stesso punto in cui era sospeso, permettendo di gestirlo se vuole, **ma non siamo obbligati a farlo**.
Infatti, nei nostri esempi, la pipeline che abbiamo non ha blocchi `try...catch`, pertanto l'errore non verrebbe propagato opportunamente.

Potrei però volere che la pipeline non veda proprio l'errore, in questo caso potrei semplicemente passare dei valori di fallback:

```
    return Promise.resolve(value).then(
      res => step(res),
      err => step(0)
    );
```

Questa modifica è completamente invisibile per la pipeline.
Pertanto, in una pipeline fatta tramite `generator/yield`

- L'errore non interrompe obbligatoriamente l'esecuzione
- Puoi reiniettare valori all'interno di essa
- Puoi cambiare politiche senza toccare la pipeline: nel caso di pipeline completamente pura, è possibile comporre senza problemi diverse pipeline in una sola ed usarle attraverso un opportuno **scheduler**
- Lo scheduler decide cosa fare: **ho il controllo totale del flusso della sua esecuzione senza intaccarne obbligatoriamente la logica applicativa**

4.5.2 Retry system

Segue da quanto visto precedentemente che basta veramente poco per parametrizzare un opportuno **scheduler** per permettere il retry delle operazioni della pipeline:

```
function* pipeline() {
  const t = yield readTemperature();
  const h = yield readHumidity();
  const p = yield readPressure();
  return { t, h, p };
}

function runWithRetry(gen, attempts = 3) {
  const it = gen();

  function execute(promise, attemptsLeft) {
    return promise.catch(err => {
      if (attemptsLeft == 0) throw err;
      return execute(promise, attemptsLeft - 1);
    });
  }

  function step(lastValue) {
    const { value, done } = it.next(lastValue);

    if (done) return Promise.resolve(value);

    return execute(value, attempts)
      .then(res => step(res))
      .catch(err => it.throw(err));
  }

  return step();
}

runWithRetry(pipeline);
```

Questo **scheduler**, per esempio, ri-esegue fino a un max di ***attempts*** volte ogni risultato su cui il generatore fa **yield**.
Tutto questo senza modificare la pipeline di partenza.

Capitolo 5

Valutazioni

La valutazione dell'efficacia dei paradigmi (e librerie) per la gestione del flusso analizzati non si concentra sulle prestazioni o sul consumo di risorse, ma sulla qualità del codice prodotto a livello della sintassi stessa che si deve usare per lo sviluppo di un programma.
L'obiettivo è comprendere in che modo ciascun paradigma influisca sulla leggibilità, sulla prevedibilità, sulla naturalezza del codice, sull'espressività e sulla manutenibilità, e sulle differenze rispetto codice sincrono.

Il paradigma `async/await` ha avuto un impatto enorme sulla comunità **JavaScript** grazie alla sua semplicità d'uso.
La sintassi molto lineare creata sopra le **Promise** ha reso questo modello estremamente popolare.
Tuttavia la sua efficacia diminuisce rapidamente quando si passa da flussi lineari a scenari più complessi.
La leggibilità, che inizialmente sembra un punto di forza, si deteriora non appena il flusso richiede cancellazioni, supervisione o retry.
Il problema principale è che `async/await` non offre strumenti nativi per modellare questi concetti, ma ogni volta che si tenta di implementare un comportamento non banale, la logica applicativa viene contaminata da dettagli relativi alla gestione del flusso, come per esempio vari blocchi **try...catch** annidati.
Il codice tende quindi a degradare rapidamente in maniera proporzionale alla complessità del flusso asincrono: la mancanza di strumenti nativi per la gestione di esso porta quindi a soluzioni ad hoc, spesso difficili da estendere e modificare, opinionate anche dallo sviluppatore stesso che le implementa.
La flessibilità del modello è quindi limitata: funziona bene per sequenze lineari e imperative, ma inizia a diventare illeggibile e non naturale quando proviamo a rap-

presentare concetti come possono essere concorrenza o composizione.
Ogni componente deve gestire autonomamente errori, cancellazioni e timeout, generando duplicazione e complessità.
La prevedibilità del flusso è ulteriormente compromessa dal cosiddetto **Await Event Horizon** (3.2): a seguito dell'utilizzo di **await**, il controllo non appartiene più al contesto logico in cui il codice è scritto, ma viene delegato al runtime delle **Promise**, rendendo difficile ragionare su cosa accade in caso di errori o cancellazioni.
Queste problematiche, oltre al modo stesso in cui operano le **Promise** come visto nel capitolo 3.3, introducono una differenza a livello di sintassi tra ciò che il codice dovrebbe fare e ciò che realmente fa, riducendo l'efficacia complessiva del paradigma.

Il paradigma `generator/yield` offre una prospettiva completamente diversa.
A differenza di `async/await`, i generatori non nascondono il flusso di esecuzione, ma lo espongono in modo esplicito.
Ogni **yield** rappresenta un punto di sospensione chiaro, e il controllo viene delegato al chiamante.
Questo rende il codice estremamente prevedibile: non esistono salti impliciti, non esistono comportamenti nascosti.
La logica applicativa può essere scritta in modo lineare, mentre la logica di controllo, come scheduling, retry, gestione errori e concorrenza, può essere completamente separata e gestita esternamente.
Questa separazione delle responsabilità è uno dei punti di forza più significativi dei generatori, e rappresenta un miglioramento sostanziale rispetto a `async/await`.
Dal punto di vista dell'espressività, i generatori sono il paradigma più potente disponibile in **JavaScript**.
La possibilità di reiniettare valori o errori all'interno del iteratore consente di implementare retry, backoff, supervisione e cancellazione in modo naturale, senza dover ricorrere a strutture artificiali (siccome ci basiamo sulla funzionalità nativa degli iteratori).
Tuttavia questa potenza ha un costo: l'uso diretto dei generatori richiede la costruzione di uno **scheduler**, ossia di un componente che gestisca la progressione del iteratore, la propagazione degli errori e, nel nostro caso, anche la risoluzione delle **Promise**.
Questo introduce complessità infrastrutturale che, pur essendo concettualmente elegante, può risultare onerosa nell'uso in ogni contesto.
Infine, anche utilizzando i generatori, non abbiamo comunque eliminato le **Promise**: esse rimangono la base del modello asincrono di **JavaScript**, e devono essere gestite, seppur in modo più localizzato e controllato.

La libreria `Effection` rappresenta un approccio radicalmente diverso e particolarmente interessante.
A differenza di `async/await`, `Effection` non utilizza direttamente le **Promise** per modellare l'asincronia, ma le usa internamente solo nel suo runtime.
Inoltre, in constrasto con il paradigma `generator/yield`, `Effection` non richiede la costruzione di uno **scheduler** o di un'infrastruttura di controllo.
Il suo modello si basa interamente sulle funzioni generatore, sfruttate come meccanismo nativo per implementare la concorrenza strutturata.
Questo significa che **Effection** è immune al problema dell'**Await Even Horizon**, siccome non esiste alcuna delega al runtime delle **Promise**, non esiste alcun salto implicito del flusso, non esiste alcuna perdita di contesto.
Il flusso asincrono rimane sempre sotto il controllo del codice che lo definisce, e la struttura del programma riflette esattamente la struttura del flusso.
`Effection` permette quindi di scrivere codice asincrono quindi che ha un aspetto identico a quello del codice sincrono.
Ogni operazione asincrona è rappresentata da un **yield**, che diventa il punto di sospensione esplicito, leggibile e prevedibile, così come per `async/await`, ma non essendoci **Promise**, non esistono callback nascoste, salti di contesto o comportamenti inattesi. Il codice è naturale, lineare e facile da seguire.
`Effection` rappresenta un miglioramento significativo rispetto sia a `async/await` sia ai generatori puri.
La struttura del codice è chiara, modulare e coerente.
Ogni componente ha un ciclo di vita ben definito, e la concorrenza è sempre confinata all'interno di scope espliciti.
Questo riduce il rischio di causare problematiche come memory leak e race condition, e tutto questo anche solo grazie a una migliore leggibilità.

Capitolo 6

Conclusioni

In **JavaScript**, le **Promise** e **async/await** hanno rappresentato un passo avanti significativo nella scrittura del codice asincrono, ma non hanno risolto alcuni limiti strutturali che emergono soprattutto nelle applicazioni moderne, dove la gestione del ciclo di vita delle operazioni è cruciale.
L'analisi del paradigma `generator/yield` ha mostrato che il linguaggio possiede già parzialmente degli strumenti per superare tali limiti: tuttavia, la necessità di dover agire manualmente sugli iteratori rende questo paradigma potente e granulare ma poco pratico per l'uso in ogni contesto.
In questo contesto si inserisce `Effection`, che introduce un runtime che sfrutta le funzioni generatore per portare in **JavaScript** la concorrenza strutturata. `Effection` mette in luce il potenziale già presente nelle funzioni generatore, offrendo un modello asincrono che permette di non utilizzare direttamente né **Promise** né **await**, ma che può comunque integrarsi con essi senza richiedere modifiche a codebase già esistenti.
Pur non eliminando la complessità intrinseca dei generatori, `Effection` dimostra quanto questo approccio possa rendere la modellazione del codice asincrono più prevedibile, leggibile e coerente, simile alla programmazione sincrona.
Proprio questa dimostrazione di potenzialità lascia intravedere una direzione futura: la possibilità di sviluppare un nuovo paradigma nel linguaggio che mantenga l'eleganza, semplicità e la prevedibilità della concorrenza strutturata di `Effection`, con la granularità dei generatori ma senza la loro complessità.
In questo senso, né `Effection` né `generator/yield` sono da considerarsi punti di arrivo, ma uno spunto per l'evoluzione dei modelli asincroni in **JavaScript**.

Bibliografia

[1] Manohar Batra. Js — generator functions vs async/await https://medium.com/@contactmanoharbatra/generator-functions-vs-async-await-24270dc7bee0. 2025.

[2] Maciej Cieślar. Implementing async and await with generators https://www.freecodecamp.org/news/how-to-implement-async-and-await-with-generators-11ab0859010f/. 2018.

[3] Paul Cowan. Javascript generators: The superior async/await https://blog.logrocket.com/javascript-generators-the-superior-async-await/. 2021.

[4] Ecma International. Ecmascript® 2015 language specification http://262.ecma-international.org/6.0/. 2015.

[5] Ecma International. Ecmascript® 2017 language specification https://262.ecma-international.org/8.0/. 2017.

[6] Lakshika. Goodbye async/await — meet the new javascript feature that makes concurrency effortless! https://javascript.plainenglish.io/goodbye-async-await-meet-the-new-javascript-feature-that-makes-concurrency-ef 2025.

[7] Taras Mankovski. Why javascript needs structured concurrency https://frontside.com/effection/blog/2026-02-06-structured-concurrency-for-javascript/. 2026.

[8] Veerendra Singh Rajput. Compare async/await and generators usage to achieve same functionality https://www.geeksforgeeks.org/javascript/compare-async-await-and-generators-usage-to-achieve-same-functionality/. 2025.

[9] XJavascript.com. Async/await vs es6 yield with generators: Key differences and implementation explained https://www.xjavascript.com/blog/

difference-between-async-await-and-es6-yield-with-generators/. 2025.