Resurse suplimentare
pe care recomand să le parcurgi

Ce sunt Promises?

Un **Promise** este un obiect care reprezintă finalizarea sau eșecul eventual al unei operațiuni asincrone. În esență, este o "promisiune" că o anumită operațiune va returna un rezultat la un moment dat în viitor.

Un Promise poate fi în una din următoarele stări:

- **Pending**: Starea inițială, nici finalizat, nici respins
- **Fulfilled**: Operațiunea s-a încheiat cu succes
- **Rejected**: Operațiunea a eșuat
- **Settled**: Promisiunea a fost fie îndeplinită, fie respinsă (nu mai este în starea pending)

Crearea unui Promise

Sintaxa de bază pentru crearea unui Promise:

```
const myPromise = new Promise((resolve, reject) => {
  // Cod asincron...

  // Dacă operațiunea reușește:
  resolve(valoare) // trece Promise-ul în starea "fulfilled"

  // Dacă operațiunea eșuează:
  reject(eroare) // trece Promise-ul în starea "rejected"
})
```

Exemplu concret:

```
function waitThreeSeconds() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Operațiune finalizată!')
    }, 3000)
  })
}

console.log('Începe operațiunea...')
waitThreeSeconds().then((result) => {
  console.log(result)
})
console.log('Promisiunea a fost creată!')
```

Output:

```
Începe operațiunea...
Promisiunea a fost creată!
Operațiune finalizată! (după 3 secunde)
```

Utilizarea Promises: .then(), .catch() și .finally()

Promises oferă metode pentru a gestiona rezultatul (sau eșecul) operațiunilor asincrone:

- **.then()**: Se execută când promisiunea este îndeplinită (resolved)
- **.catch()**: Se execută când promisiunea este respinsă (rejected)
- **.finally()**: Se execută indiferent de rezultatul promisiunii

```javascript
function fetchUserData(id) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (id <= 0) {
        reject(new Error('ID invalid'))
      } else {
        const user = {
          id: id,
          nume: 'Ion Popescu',
          email: 'ion@exemplu.ro',
        }
        resolve(user)
      }
    }, 1500)
  })
}

fetchUserData(123)
  .then((user) => {
    console.log('Utilizator găsit:', user)
  })
  .catch((error) => {
    console.error('A apărut o eroare:', error.message)
  })
  .finally(() => {
    console.log('Operațiunea s-a încheiat (cu succes sau eșec)')
  })
```

Transformarea callback-urilor în Promises

Putem transforma orice funcție bazată pe callback-uri într-una bazată pe Promise:

```javascript
// Funcție bazată pe callback
function readFileWithCallback(path, callback) {
  setTimeout(() => {
    if (path.endsWith('.txt')) {
      callback(null, 'Conținutul fișierului text')
    } else {
      callback(new Error('Format de fișier nesuportat'))
    }
  }, 1000)
}

// Transformată în Promise
function readFile(path) {
  return new Promise((resolve, reject) => {
    readFileWithCallback(path, (error, content) => {
      if (error) {
        reject(error)
      } else {
        resolve(content)
      }
    })
  })
}

// Utilizare
readFile('document.txt')
  .then((content) => {
    console.log('Conținut:', content)
  })
  .catch((error) => {
    console.error('Eroare:', error.message)
  })
```

Înlănțuirea Promises

Unul dintre cele mai mari avantaje ale Promises este capacitatea de a fi înlănțuite, ceea ce ne permite să scriem secvențe de operațiuni asincrone într-un mod clar și lizibil:

GENERAȚIA TECH

```
function getUser(id) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (id > 0) {
        resolve({ id: id, nume: 'Ana Ionescu' })
      } else {
        reject(new Error('ID utilizator invalid'))
      }
    }, 1000)
  })
}

function getUserOrders(user) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (user && user.id) {
        resolve({
          user: user,
          orders: [
            { id: 1, produs: 'Laptop', pret: 3500 },
            { id: 2, produs: 'Telefon', pret: 2000 },
          ],
        })
      } else {
        reject(new Error('Utilizator invalid'))
      }
    }, 1000)
  })
}

function calculateTotal(data) {
  return new Promise((resolve) => {
    setTimeout(() => {
      const total = data.orders.reduce((sum, order) => sum + order.pret, 0)
      resolve({
        ...data,
        total: total,
      })
    }, 500)
  })
}

// Utilizare înlănțuită
getUser(123)
  .then((user) => {
    console.log('Utilizator găsit:', user)
    return getUserOrders(user)
  })
  .then((data) => {
    console.log(`${data.user.nume} are ${data.orders.length} comenzi`)
    return calculateTotal(data)
  })
  .then((finalResult) => {
    console.log(`Totalul comenzilor: ${finalResult.total} lei`)
  })
  .catch((error) => {
    console.error('Eroare în proces:', error.message)
  })
```

Observă cum fiecare `.then()` returnează un nou Promise, permițând înlănțuirea elegantă a operațiunilor.

Promise.all(): Procesarea paralelă a Promises

Când avem mai multe operațiuni asincrone independente care trebuie executate, putem utiliza `Promise.all()` pentru a le rula în paralel și a aștepta finalizarea tuturor:

```
function getUsers() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(['Ana', 'Ion', 'Maria'])
    }, 2000)
  })
}

function getProducts() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(['Laptop', 'Telefon', 'Tabletă'])
    }, 1500)
  })
}

function getCategories() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(['Electronice', 'Îmbrăcăminte', 'Cărți'])
    }, 1000)
  })
}

console.log('Începe încărcarea datelor...')

Promise.all([getUsers(), getProducts(), getCategories()])
  .then(([users, products, categories]) => {
    console.log('Toți utilizatorii:', users)
    console.log('Toate produsele:', products)
    console.log('Toate categoriile:', categories)
    console.log('Toate datele au fost încărcate!')
  })
  .catch((error) => {
    console.error('Una din operațiuni a eșuat:', error)
  })
```

`Promise.all()` returnează un nou Promise care se rezolvă când toate Promises din array s-au rezolvat, sau se respinge dacă oricare dintre ele eșuează.

Alte metode utile pentru Promises

Promise.race()

Returnează primul Promise care se rezolvă sau se respinge:

```
const promise1 = new Promise((resolve) =>
  setTimeout(() => resolve('Primul'), 500)
)
const promise2 = new Promise((resolve) =>
  setTimeout(() => resolve('Al doilea'), 100)
)

Promise.race([promise1, promise2]).then((winner) => {
  console.log('Câștigătorul este:', winner) // Va afișa "Al doilea"
})
```

Promise.allSettled()

Așteaptă ca toate Promises să fie finalizate (resolved sau rejected) și returnează un array cu rezultatele lor:

```
const promises = [
  Promise.resolve('Succes'),
  Promise.reject('Eșec'),
  Promise.resolve('Alt succes'),
]

Promise.allSettled(promises).then((results) => {
  console.log(results)
  // Rezultate va fi un array cu obiecte de forma:
  // [
  //   { status: "fulfilled", value: "Succes" },
  //   { status: "rejected", reason: "Eșec" },
  //   { status: "fulfilled", value: "Alt succes" }
  // ]
})
```

Promise.any()

Returnează primul Promise care se rezolvă cu succes (și ignoră eșecurile):

```
const promises = [
  new Promise((resolve, reject) =>
    setTimeout(() => reject(new Error('Eșec 1')), 100)
  ),
  new Promise((resolve) => setTimeout(() => resolve('Succes'), 200)),
  new Promise((resolve) => setTimeout(() => resolve('Alt succes'), 300)),
]

Promise.any(promises)
  .then((first) => {
    console.log('Primul succes:', first) // Va afișa "Succes"
  })
  .catch((error) => {
    console.error('Toate au eșuat:', error)
  })
```

Gestionarea erorilor în Promises

Erorile în lanțuri de Promises sunt propagate până la primul `.catch()`:

```
getUser(-1) // Va eșua cu "ID utilizator invalid"
  .then((user) => {
    console.log('Utilizator găsit:', user) // Nu se va executa
    return getUserOrders(user)
  })
  .then((data) => {
    console.log('Date comenzi:', data) // Nu se va executa
    return calculateTotal(data)
  })
  .catch((error) => {
    console.error('A apărut o eroare:', error.message) // Va prinde eroarea
    return { error: true, mesaj: error.message } // Returnează un obiect pentru a continu
a lanțul
  })
  .then((result) => {
    console.log('Rezultat final:', result) // Va afișa obiectul de eroare
  })
```

Este important să adăugăm întotdeauna un `.catch()` la lanțurile de Promises pentru a gestiona erorile potențiale.