Технічні науки

УДК 004.5

Кхатер Філіп Еліас

аспірант факультету кібернетики

Міжнародного економіко-гуманітарного університету

імені академіка Степана Дем'янчука

Kkhater Filip Elias

Postgraduate Student of the Faculty of Cybernetics

Academician Stepan Demianchuk International University of

Economics and Humanities

Науковий керівник:

Джунь Йосип Володимирович

доктор фізико-математичних наук, професор

Міжнародний економіко-гуманітарний університет

імені академіка Степана Дем'янчука

ПРОМІСИ У Node.js: ОГЛЯД І ВИКОРИСТАННЯ
PROMISES IN Node.js: OVERVIEW AND USAGE

Анотація. *У цій статті розглянуто концепцію промісів у Node.js, їх основні функції та переваги у порівнянні з традиційними зворотними викликами. Проміси були створені для вирішення низки проблем, що виникають при роботі з асинхронними операціями, таких як "callback hell" і складність обробки помилок. Стаття надає огляд основних методів роботи з промісами та демонструє приклади їх використання у Node.js. Завершується стаття висновками про ефективність промісів і їх важливість у сучасному програмуванні на JavaScript.*

Ключові слова: *проміси, Node.js, асинхронне програмування, зворотні виклики, callback hell, обробка помилок, JavaScript.*

Summary. *This article examines the concept of promises in Node.js, their main functions, and advantages compared to traditional callbacks. Promises were created to address a number of issues that arise when working with asynchronous operations, such as "callback hell" and the complexity of error handling. The article provides an overview of the main methods for working with promises and demonstrates examples of their use in Node.js. The article concludes with insights on the effectiveness of promises and their importance in modern JavaScript programming.*

Key words: *promises, Node.js, asynchronous programming, callbacks, callback hell, error handling, JavaScript.*

Node.js є популярною платформою для створення серверних додатків, що використовує асинхронну модель вводу-виводу для досягнення високої продуктивності. Одним з ключових аспектів роботи з асинхронними операціями в Node.js є управління ними за допомогою промісів (Promises). Проміси дозволяють писати асинхронний код, який є більш читабельним, зручним і легким для підтримки в порівнянні з традиційними зворотними викликами (callbacks). У цій статті ми розглянемо, для чого були створені проміси, їх основні принципи роботи, та приклади їх використання у Node.js.

Мета статті полягає, щоб показати роль промісів і які проблеми були вирішенні. Однією з основних проблем при роботі з асинхронними операціями у JavaScript є так званий "callback hell". Це явище виникає, коли асинхронні функції вкладаються одна в одну, що призводить до коду з глибокою вкладеністю та низькою читабельністю. Це ускладнює відстеження потоку

виконання програми і робить її складною для підтримки. При використанні зворотних викликів обробка помилок стає складною, оскільки необхідно перевіряти наявність помилок на кожному рівні вкладеності. Це призводить до дублювання коду та збільшує ймовірність пропуску помилок.Зворотні виклики не надають зручних механізмів для роботи з кількома асинхронними операціями одночасно. Це ускладнює об'єднання результатів кількох операцій і може призвести до неконтрольованого росту складності коду.Асинхронний код на основі зворотних викликів часто виглядає заплутаним і важким для підтримки, особливо у великих проектах. Проміси дозволяють писати код у більш декларативному стилі, що робить його більш читабельним і зрозумілим.

Проміс (Promise) — це об'єкт, який представляє завершення або невдачу асинхронної операції та її результат. Проміси були введені для полегшення роботи з асинхронним кодом і для заміни класичного підходу на основі зворотних викликів.

Проміс може перебувати в одному з трьох станів:

- Очікування (pending) — початковий стан, у якому проміс очікує завершення асинхронної операції.
- Виконано (fulfilled) — стан, у якому асинхронна операція завершилася успішно, і проміс має результат.
- Відхилено (rejected) — стан, у якому асинхронна операція завершилася з помилкою, і проміс має причину відхилення.

Основні Методи Промісів

- then(): використовується для визначення дій, які будуть виконані після успішного завершення промісу.
- catch(): використовується для обробки помилок, що виникають під час виконання промісу.

- finally(): виконується після завершення промісу незалежно від його результату.

```
const myPromise = new Promise((resolve, reject) => {
  const success = true; // або false в залежності від умови
  if (success) {
    resolve('Операція успішна');
  } else {
    reject('Сталася помилка');
  }
});

myPromise
  .then(result => {
    console.log(result); // Виведе 'Операція успішна' у разі успіху
  })
  .catch(error => {
    console.error(error); // Виведе 'Сталася помилка' у разі невдачі
  });
```

Рис. 1. Приклад створення Промісу

Методи Promise.all та Promise.race дозволяють працювати з кількома промісами одночасно.

```
const promise1 = Promise.resolve(3);
const promise2 = 42;
const promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([promise1, promise2, promise3])
  .then(values => {
    console.log(values); // [3, 42, 'foo']
  })
  .catch(error => console.error(error));
```

Рис. 2. Приклад запуску одночасно декількох промісів

Проміси мають вбудовані методи для обробки помилок (.catch()), що дозволяє централізовано обробляти помилки в одному місці. Порівняння двох підходів обробки помилок через зворотній виклик та обробку через вбудовані функції промісів.

```
// Обробка помилок у зворотних викликах
doSomething(function(err, result) {
  if (err) {
    console.error(err);
  } else {
    doSomethingElse(result, function(err, newResult) {
      if (err) {
        console.error(err);
      } else {
        doThirdThing(newResult, function(err, finalResult) {
          if (err) {
            console.error(err);
          } else {
            console.log(finalResult);
          }
        });
      }
    });
  }
});

// Обробка помилок у промісах
doSomething()
  .then(result => doSomethingElse(result))
  .then(newResult => doThirdThing(newResult))
  .then(finalResult => console.log(finalResult))
  .catch(error => console.error(error));
```

Рис. 3. Приклад обробки помилок у зворотних викликах та промісах

В прикладі [3] видно як за допомогою промісів легко структурувати код та обробляти помилки.

Отже, проміси у Node.js вирішують кілька важливих проблем асинхронного програмування, таких як "callback hell", складність обробки помилок, керування множинними асинхронними операціями та покращення читабельності коду. Вони забезпечують більш зручний та ефективний спосіб роботи з асинхронним кодом, що робить їх важливим інструментом для будь-якого розробника JavaScript. Використання промісів дозволяє створювати більш надійні та масштабовані додатки, що є важливою перевагою у сучасному програмуванні.

Література

1. Node.js Documentation. URL: https://nodejs.org/en/docs/ (дата звернення: 20.05.2024).

2. MDN Web Docs on Promises. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise (дата звернення: 10.05.2024).

3. Flanagan D. JavaScript: The Definitive Guide. O'Reilly Media, 2020.

4. Osmani A. Learning JavaScript Design Patterns. O'Reilly Media, 2014.

5. Ferguson R. Practical Node.js: Building Real-World Scalable Web Apps. Apress, 2018.