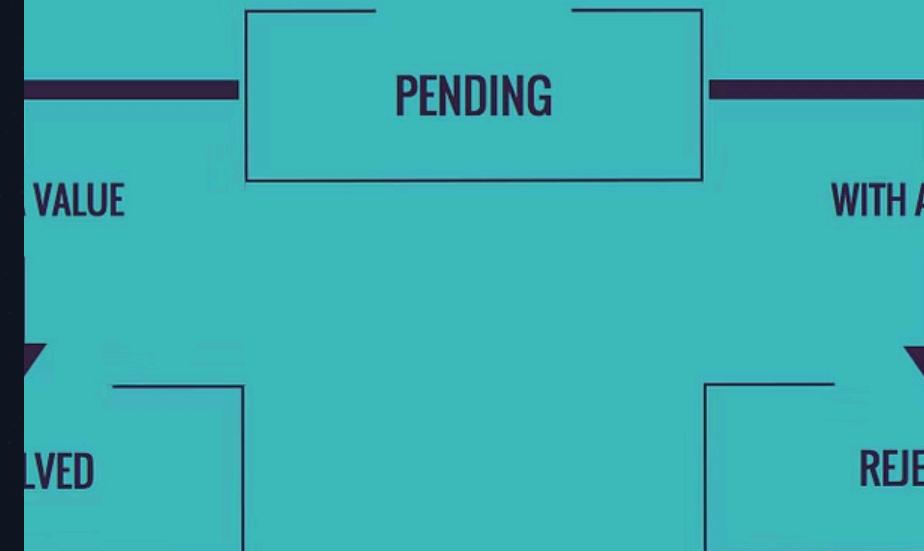


Обробка асинхронних операцій за допомогою Promise

Розуміння Promise — це ключ до ефективної роботи з асинхронним кодом у JavaScript. Дізнайтесь, як керувати асинхронними операціями елегантно та передбачувано.

PROMISES

JAVASCRIPT



Що таке Promise

Концепція

Promise — це об'єкт, що представляє результат виконання асинхронної операції, яка може завершитися успішно або з помилкою. На відміну від колбеків, проміси забезпечують кращий та більш зрозуміліший спосіб роботи з асинхронними операціями.

```
// Callback
asyncFunction(a, b, result => {
  console.log(result);
});

// Promise
asyncFunction(a, b)
  .then(result => {
    console.log(result);
  });
}
```

Переваги Promise

- **Відсутність інверсії керування:** функції повертають результати, контроль залишається за викликачем
- **Простіший ланцюжок:** методи then() об'єднуються у зручний ланцюжок
- **Поседнання асинхронних викликів:** працюємо з даними як з об'єктами
- **Обробка помилок:** виключення та асинхронні помилки обробляються однаково

З чого складається Promise

pending

Початковий стан на момент створення Promise

fulfilled

Успішне завершення операції з результатом

rejected

Завершення з помилкою

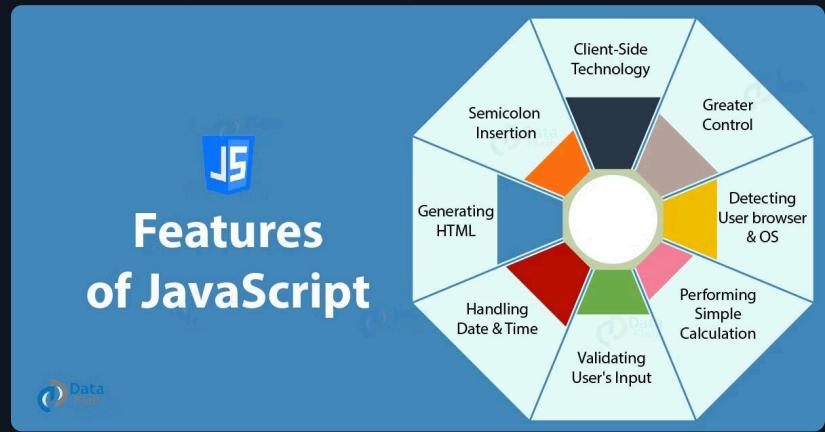
Коли створюється новий проміс через `new Promise`, йому передається функція **executor**, яка запускається автоматично. Вона містить код асинхронної операції та отримує два колбеки: `resolve` і `reject`.

```
new Promise(function(resolve, reject) {  
  ...  
  if (...) {  
    resolve(value); // success  
  } else {  
    reject(reason); // failure  
  }  
});
```

- **Важливо:** Стан Promise після завершення є незмінним. Promise може бути виконаний лише один раз, наступні спроби не мають ефекту.

state: pending

"Споживання" промісу



Методи обробки результату

Як споживач Promise, ви отримуєте повідомлення про виконання або відхилення через реакції — колбеки, які реєструєте за допомогою методів `then()` і `catch()`:

`then()`

Приймає дві функції: перша для успішного завершення, друга — для помилки

```
promise.then(  
  value => { /* fulfillment */ },  
  error => { /* rejection */ }  
)
```

`catch()`

Спрацьовує виключно при помилці. Зручніша та рекомендована альтернатива

```
promise.catch(  
  error => { /* rejection */ }  
)
```

Проміси дуже корисні для асинхронних функцій з одноразовим результатом. Після виконання проміс більше не змінюється, тому не виникає race condition — не має значення, чи викликаєте `then()` або `catch()` до або після виконання.

catch() та finally()

1

catch() особливості

Спрацьовує при відхиленні (reject), але так само, як і then(), перетворює дії колбека на Promise. Значення, повернуте функцією зворотного виклику, стає значенням виконання:

```
new Promise(_, reject) =>
  reject(new Error('My error'))
)
.catch(() => 'default value')
.then(console.log);
```

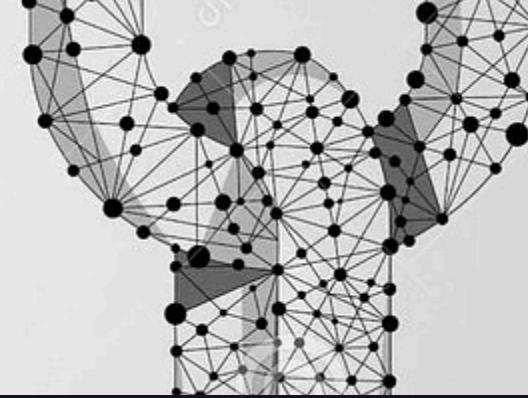
2

finally() застосування

Приймає колбек без аргументів. Виконується завжди, незалежно від статусу Promise і не впливає на значення, що повертаються then() та catch().

```
new Promise(resolve =>
  resolve('fulfilled')
)
.then(result => {
  console.log(`then ${result}`);
})
.catch(error => {
  console.log(`catch ${error}`);
})
.finally(() => {
  console.log('finally');
})
```

Зазвичай finally() використовують наприкінці для виконання дій незалежно від результату — наприклад, прибрати індикатор завантаження файлу.



Ланцюг споживачів

Оскільки `then()` та `catch()` завжди повертають `Promise`, можна створювати довгі ланцюжки викликів методів:

```
function asyncFunc() {
  return asyncFuncA()
    .then(result => {
      // ...
      return asyncFuncB();
    })
    .then(result => {
      // ...
      return result;
    })
    .then(result => {
      // ...
      return asyncFuncC();
    });
}
```

Послідовне виконання

У якомусь сенсі `then()` є асинхронною версією синхронної крапки з комою, бо виконує дві асинхронні операції послідовно.

Централізована обробка помилок

Можна додати `catch()` і дозволити йому обробляти декілька джерел помилок одночасно:

```
asyncFuncA()
  .then(result => {
    return asyncFunctionB();
  })
  .then(result => {
    // ...
  })
  .catch(error => {
    // Обробка помилок asyncFuncA(),
    // asyncFuncB() та будь-які виключення
 });
```

Також ви можете обробляти помилки після кожного виклику `then()` для більш детального контролю.

Необроблені помилки

01

Виникнення проблеми

Може виникнути ситуація з необробленою помилкою — наприклад, забули додати `catch()` в кінець ланцюжка

02

Пошук обробника

При помилці виконання переходить до найближчого обробника помилок. Якщо обробника немає, помилка «застряє»

03

Подія `unhandledrejection`

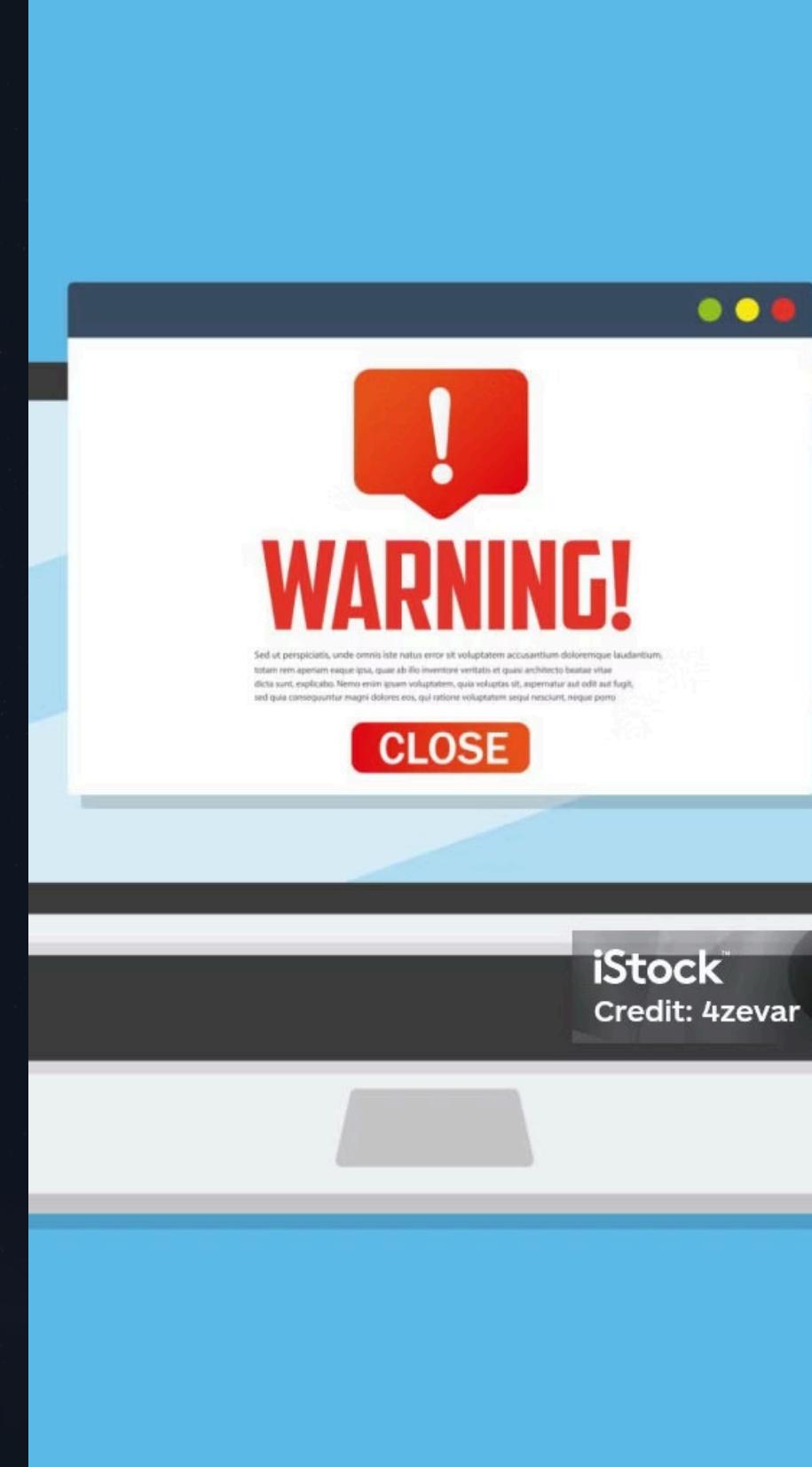
Генерується подія `unhandledrejection`, і відповідний об'єкт `event` містить інформацію про помилку

04

Глобальна помилка

Без обробника події движок генерує глобальну помилку, повідомлення про яку відображається в консолі

- Висновок:** Завжди пам'ятайте про обробники помилок! Добавайте `catch()` в кінець ланцюжків `Promise` для запобігання необробленим помилкам.



Promise API: resolve та reject

Promise.resolve()

Статичний метод для створення вже виконаного (fulfilled) проміса з певним значенням. Корисний для початку асинхронного ланцюжка.

```
Promise.resolve(value);
```

Це коротший запис:

```
new Promise((resolve) =>  
    resolve(value)  
)
```

Приклад використання:

```
Promise  
.resolve(42)  
.then(result => {  
    console.log(result); // 42  
});
```

Дозволяє перетворювати синхронні значення на асинхронні, що забезпечує однаковість у обробці Promise-ланцюжків.

Promise.reject()

Метод для створення вже відхиленого (rejected) промісу з певною причиною. Дозволяє явно зазначити помилку.

```
Promise.reject(reason);
```

Це коротший запис:

```
new Promise((resolve, reject) =>  
    reject(error)  
)
```

Приклад використання:

```
Promise  
.reject("Something went wrong")  
.catch(error => {  
    console.error(error);  
    // Something went wrong  
});
```

Використовується рідко, але корисний для явного позначення помилки в асинхронній операції.

Promise.all()

Статичний метод для створення Promise, який очікує на виконання **всіх** переданих Promise і повертає масив з їх результатами в тому ж порядку.

Синтаксис

```
Promise.all(iterable);
```

Приймає ітерабельний об'єкт (зазвичай масив) з Promise

Поведінка при успіху

Promise-результат буде resolved тільки коли **кожен** Promise завершиться успішно. Результат — масив значень у порядку передачі.

Поведінка при помилці

Якщо **хоча б один** Promise завершується з помилкою, Promise.all() негайно відхиляється без очікування інших.

Успішне виконання:

```
const promiseA = Promise.resolve('Hello');
const promiseB = Promise.resolve('Promise');
const promiseC = Promise.resolve('All');

const collection = Promise.all([
  promiseA, promiseB, promiseC
]);

collection
  .then(x => x.toString()
    .replaceAll(',', ' '))
  .then(console.log);
// Hello Promise All
```

Відхилення при помилці:

```
const a = Promise.resolve('Hello');
const b = Promise.reject('Rejected');
const c = Promise.resolve('All');

const collection = Promise.all([
  a, b, c
]);

collection
  .then(x => x.toString()
    .replaceAll(',', ' '))
  .then(console.log)
  .catch(console.log);
// Rejected
```

Promise API: allSettled, race, any

Promise.allSettled()

Очікує завершення **всіх** Promise і повертає масив об'єктів зі статусом кожного (fulfilled/rejected). На відміну від all(), завжди повертає результат.

```
Promise.allSettled([promiseA,  
promiseB])  
.then(console.log);  
// [{status: 'fulfilled', value: ...},  
// {status: 'rejected', reason: ...}]
```

Promise.race()

Виконується, як тільки **перший** із переданих Promise завершиться (успішно чи з помилкою). Повертає результат або помилку першого завершеного.

```
Promise.race([promiseA,  
promiseB])  
.then(x => console.log('First:',  
x))  
.catch(x => console.log('Error:',  
x));
```

Promise.any()

Виконується, як тільки хоча **б** **один** Promise завершиться **успішно**. Повертає перший успішний результат. Якщо всі відхилені — повертає AggregateError.

```
Promise.any([promiseA, promiseB])  
.then(x => console.log('Success:', x))  
.catch(x => console.log('All failed'));
```

Standard Lorem Ipsum pass



ipsum dolor sit amet, consectetur adipiscing elit, sed do
tempor incididunt ut labore et dolore magna aliqua.

a
[Learn more](#)