1. Обращение к API и получение данных: Разработчики используют Promises для выполнения HTTP-запросов, чтобы получать данные с удаленного сервера. Например, fetch API возвращает Promise, который разрешается с результатом запроса.

fetch('https://api.example.com/data')

.then(response => response.json())

.then(data => console.log(data))

.catch(error => console.error('Ошибка:', error));

1. Последовательное выполнение: Когда одна асинхронная операция зависит от другой, Promises используются для упорядочивания выполнения функций.

asyncOperation1()

.then(result1 => asyncOperation2(result1))

.then(result2 => console.log('Результат:', result2))

.catch(error => console.error('Ошибка:', error));

1. Синхронизация множественных асинхронных операций: Promises могут использоваться для параллельного выполнения нескольких асинхронных задач и ожидания их завершения с помощью Promise.all.

const promise1 = fetch('https://api.example.com/data1');

const promise2 = fetch('https://api.example.com/data2');

Promise.all([promise1, promise2])

.then(responses => Promise.all(responses.map(r => r.json())))

.then(data => console.log('Данные:', data))

.catch(error => console.error('Ошибка:', error));

1. Таймеры и задержки: Promises могут быть использованы для создания задержки выполнения кода, что полезно при тестировании или для реализации троттлинга API запросов.

function delay(ms) {

return new Promise(resolve => setTimeout(resolve, ms));

}

delay(2000).then(() => console.log('2 секунды спустя'));

1. Анимации: В браузерах Promises могут быть использованы для упрощения работы с CSS-анимациями или JavaScript-анимациями, ожидая завершения анимации.

function animateElement(element) {

return new Promise(resolve => {

element.addEventListener('animationend', resolve, { once: true });

element.classList.add('animate');

});

}

animateElement(myElement).then(() => console.log('Анимация завершена'));

1. Выравнивание асинхронных операций: Promises часто используемые при реализации сложных сценариев потоков данных, где обработка результатов одной задачи инициирует другие задачи.

const fetchData = () => Promise.resolve('данные');

fetchData()

.then(data => processData(data))

.then(processedData => saveData(processedData))

.then(() => console.log('Все операции завершены'))

.catch(error => console.error('Ошибка в процессе:', error));

1. Очереди задач: Promises могут быть полезны для реализации очередей задач, где каждая задача выполняется асинхронно.

const taskQueue = [task1, task2, task3];

taskQueue.reduce((promise, task) =>

promise.then(() => task()), Promise.resolve()

).then(() => console.log('Все задачи выполнены'));

1. Пользовательские события: Promises могут использоваться для того, чтобы ждать пользовательских событий, таких как нажатие кнопки.

function waitForClick(button) {

return new Promise(resolve => {

button.addEventListener('click', resolve, { once: true });

});

}

waitForClick(document.getElementById('myButton'))

.then(() => console.log('Кнопка нажата!'));

1. Проверка состояния системы: Использование Promises для асинхронных проверок статуса системы или компонента перед выполнением операций.

function checkSystemReady() {

return new Promise((resolve, reject) => {

const isReady = true; // некая проверка готовности системы

if (isReady) {

resolve('Система готова');

} else {

reject('Система не готова');

}

});

}

checkSystemReady()

.then(status => console.log(status))

.catch(error => console.error(error));

1. Перехват и повторение неудачных запросов: Promises можно использовать для реализации механизма повторной отправки неудавшихся запросов, что важно в условиях ненадежных сетей.

const fetchWithRetry = (url, retries) => {

return fetch(url).catch(error => {

if (retries > 0) {

return fetchWithRetry(url, retries - 1);

}

throw error;

});

};

fetchWithRetry('https://api.example.com/data', 3)

.then(response => response.json())

.then(data => console.log(data))

.catch(error => console.error('Запрос не удался:', error));

1. Интерактивные пользовательские интерфейсы: Promises могут помочь в асинхронной обработке ввода пользователя, например, при валидации данных форм. Это обеспечивает плавную и отзывчивую работу интерфейса.

function validateInput(input) {

return new Promise((resolve, reject) => {

if (input.length > 3) {

resolve('Ввод корректен');

} else {

reject('Слишком короткий ввод');

}

});

}

document.getElementById('inputField').addEventListener('input', (event) => {

validateInput(event.target.value)

.then(message => console.log(message))

.catch(error => console.error(error));

});

1. Контроль над последовательностью вызовов: Управление порядком выполнения функций, которые зависят от асинхронных данных.

function step1() { return Promise.resolve('Шаг 1 завершен'); }

function step2() { return Promise.resolve('Шаг 2 завершен'); }

step1()

.then(result => {

console.log(result);

return step2();

})

.then(result => console.log(result));

1. Использование fetch API для загрузки данных

Пример из реального приложения веб-сайта, использующего API для получения данных о погоде.

function getWeather(city) {

const apiKey = 'YOUR\_API\_KEY';

const url=`https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${apiKey}`;

return fetch(url)

.then(response => {

if (!response.ok) {

throw new Error('Network response was not ok ' + response.statusText);

}

return response.json();

})

.catch(error => console.error('Ошибка:', error));

}

getWeather('Moscow').then(data => console.log(data));

1. Загрузка изображений с использованием Promises

Пример загрузки нескольких изображений, где нужно дождаться загрузки всех, прежде чем показать их на странице.

function loadImage(url) {

return new Promise((resolve, reject) => {

const img = new Image();

img.onload = () => resolve(url);

img.onerror = () => reject(new Error(`Не удалось загрузить изображение: ${url}`));

img.src = url;

});

}

const imageUrls = ['image1.jpg', 'image2.jpg', 'image3.jpg'];

Promise.all(imageUrls.map(loadImage))

.then(loadedImages => {

console.log('Все изображения загружены:', loadedImages);

// Здесь вы можете отобразить изображения на странице

})

.catch(error => console.error('Ошибка загрузки изображений:', error));

1. Инициализация пользовательских данных

Пример из приложения, где необходимо получить данные пользователя, а затем загрузить дополнительные данные, такие как список заказов.

function fetchUserData(userId) {

return fetch(`/api/users/${userId}`).then(response => response.json());

}

function fetchUserOrders(userId) {

return fetch(`/api/users/${userId}/orders`).then(response => response.json());

}

fetchUserData(1)

.then(user => {

console.log('Данные пользователя:', user);

return fetchUserOrders(user.id);

})

.then(orders => {

console.log('Заказы пользователя:', orders);

})

.catch(error => console.error('Произошла ошибка:', error));

1. Обработка форм

Пример валидации формы и отправки данных на сервер с помощью промисов.

function submitForm(data) {

return new Promise((resolve, reject) => {

fetch('/api/submit', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

},

body: JSON.stringify(data),

})

.then(response => {

if (!response.ok) {

throw new Error('Ошибка сети');

}

return response.json();

})

.then(resolve)

.catch(reject);

});

}

const formData = { name: 'Иван', email: 'ivan@example.com' };

submitForm(formData)

.then(response => console.log('Форма отправлена:', response))

.catch(error => console.error('Ошибка отправки формы:', error));

1. Фоновая обработка данных

Promises могут использоваться для выполнения фоновых задач, таких как обработка больших объемов данных, без блокировки основного потока выполнения.

function processDataInBackground(data) {

return new Promise((resolve) => {

const processedData = data.map(item => item \* 2); // Пример обработки

resolve(processedData);

});

}

processDataInBackground([1, 2, 3, 4, 5]).then(result => {

console.log('Обработанные данные:', result);

});

1. Система уведомлений

Promises можно использовать для управления всплывающими уведомлениями, когда они должны быть отображены после завершения определенных асинхронных задач.

function showNotification(message) {

return new Promise((resolve) => {

setTimeout(() => {

console.log('Уведомление:', message);

resolve();

}, 1000);

});

}

showNotification('Вы успешно вошли в систему').then(() => {

console.log('Уведомление отображено');

});

1. Параллельная обработка данных

Promises можно использовать для параллельной обработки данных из нескольких источников. Например, загрузка данных из нескольких API одновременно с помощью Promise.all.

function fetchData(url) {

return fetch(url).then(response => response.json());

}

const urls = [

'https://api.example.com/data1',

'https://api.example.com/data2',

];

Promise.all(urls.map(fetchData))

.then(results => {

console.log('Результаты:', results);

})

.catch(error => {

console.error('Ошибка при загрузке данных:', error);

});