JavaScript Promises Guide

INTRODUCTION TO PROMISES

Promises in JavaScript are objects that represent the eventual completion (or failure) of an asynchronous operation. They provide a cleaner, more powerful way to handle asynchronous code compared to traditional callbacks.

A Promise is in one of three states:
• **Pending**: Initial state, neither fulfilled nor rejected
• **Fulfilled**: The operation completed successfully
• **Rejected**: The operation failed

CREATING PROMISES

You create a Promise using the Promise constructor, which takes a function with two parameters: resolve and reject.

Example:
```
const myPromise = new Promise((resolve, reject) => {
  const success = true;
  if (success) {
    resolve("Operation successful!");
  } else {
    reject("Operation failed!");
  }
});
```

USING PROMISES WITH .THEN() AND .CATCH()

Promises are consumed using the .then() method for successful outcomes and .catch() for errors.

Example:
```
myPromise
  .then(result => {
    console.log(result); // "Operation successful!"
  })
  .catch(error => {
    console.error(error);
  });
```

PROMISE CHAINING

One of the most powerful features of Promises is the ability to chain them. Each .then() returns a new Promise, allowing you to perform sequential asynchronous operations.

Example:
```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => {
    console.log(data);
    return data.id;
  })
  .then(id => fetch(`https://api.example.com/details/${id}`))
  .then(response => response.json())
  .catch(error => console.error('Error:', error));
```

ASYNC/AWAIT SYNTAX

ES2017 introduced async/await, which provides a more synchronous-looking way to work with Promises. An async function always returns a Promise, and await pauses execution until the Promise resolves.

Example:
```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
    return data;
  } catch (error) {
    console.error('Error:', error);
  }
}
```

PROMISE.ALL() - PARALLEL EXECUTION

Promise.all() takes an array of Promises and returns a single Promise that resolves when all input Promises have resolved, or rejects if any Promise rejects.

Example:
```
const promise1 = fetch('https://api.example.com/users');
const promise2 = fetch('https://api.example.com/posts');
const promise3 = fetch('https://api.example.com/comments');

Promise.all([promise1, promise2, promise3])
  .then(responses => Promise.all(responses.map(r => r.json())))
  .then(data => {
    console.log('All data loaded:', data);
  })
  .catch(error => console.error('One or more requests failed:', error));
```

PROMISE.RACE()

Promise.race() returns a Promise that resolves or rejects as soon as one of the Promises in the array resolves or rejects.

Example:

```
const timeout = new Promise((_, reject) =>
  setTimeout(() => reject('Timeout!'), 5000)
);

Promise.race([fetch('https://api.example.com/data'), timeout])
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

ERROR HANDLING BEST PRACTICES

Always handle Promise rejections to avoid unhandled rejection warnings. Use .catch() at the end of Promise chains or try/catch with async/await.

Common Patterns:

- Always add .catch() to Promise chains
- Use try/catch blocks with async/await
- Consider using .finally() for cleanup operations
- Handle errors at appropriate levels in your application

PRACTICAL EXAMPLE: FETCHING MULTIPLE URLS

Example combining multiple concepts:

```
async function fetchMultipleUrls(urls) {
  try {
    const promises = urls.map(url => fetch(url));
    const responses = await Promise.all(promises);
    const data = await Promise.all(responses.map(r => r.json()));
    return data;
  } catch (error) {
    console.error('Failed to fetch:', error);
    throw error;
  }
}

// Usage
const urls = [
  'https://api.example.com/endpoint1',
  'https://api.example.com/endpoint2'
];

fetchMultipleUrls(urls)
  .then(results => console.log('All results:', results))
  .catch(error => console.error('Error:', error));
```