Understanding JavaScript Promises

A promise represents the eventual result of an asynchronous operation. It is a placeholder into which the successful result value or reason for failure will materialize.

Why Use Promises?

Promises provide a simpler alternative for executing, composing, and managing asynchronous operations when compared to traditional callback-based approaches. They also allow you to handle asynchronous errors using approaches that are similar to synchronous `try/catch`.

Promise States

A promise can be in one of 3 states:

- Pending - the promise's outcome hasn't yet been determined, because the asynchronous operation that will produce its result hasn't completed yet.
- Fulfilled - the asynchronous operation has completed, and the promise has a value.
- Rejected - the asynchronous operation failed, and the promise will never be fulfilled. In the rejected state, a promise has a *reason* that indicates why the operation failed.

When a promise is pending, it can transition to the fulfilled or rejected state. Once a promise is fulfilled or rejected, however, it will never transition to any other state, and its value or failure reason will not change.

Promises have been implemented in many languages, and while promise APIs differ from language to language, JavaScript promises have converged to the Promises/A+ (https://promisesaplus.com) proposed standard. EcmaScript 6 is slated to provide promises as a first-class language feature, and they will be based on the Promises/A+ proposal.

Using Promises

The primary API for a promise is its `then` method, which registers callbacks to receive either the eventual value or the reason why the promise cannot be fulfilled. Here is a simple

Related Resources

Getting Started Guides

Consuming a RESTful Web Service with jQuery (/guides/gs/consuming-rest-jquery)

Consuming a RESTful Web Service with rest.js (/guides/gs/consuming-rest-restjs)