JavaScript Promises

Ingo Köster

Diplom Informatiker (FH)

Motivation

- Für die Ausführung von JavaScript steht nur ein Thread zur Verfügung
  - Mit Ausnahme von z.B. Web Workern
- Langlaufende Operationen werden häufig asynchron ausgeführt
  - Z.B. das Lesen einer Datei, Download von Daten, etc.
- Um auf das Ende einer asynchronen Funktion reagieren zu können, wird oft eine Callback-Funktion verwendet
  - Informiert über den Erfolg oder Misserfolg der Operation
  - Im Callback steht das Resultat zur Verfügung
    - Z.B. Inhalt einer Datei

Probleme mit Callbacks

- Die Verwendung von Callbacks wird schnell unübersichtlich
- Oft stößt das Ergebnis einer asynchronen Operation eine weitere asynchrone Operation an, usw.
- Durch Einrückung wandert der Quellcode immer mehr nach rechts
  - Sog. „Pyramid of Doom“
- In der Verarbeitung können Fehler auftreten, welche ebenfalls behandelt werden müssen

Pyramid of Doom

```
function pyramidOfDoom()
{
  setTimeout(() => {
    console.log(1);
    setTimeout(() => {
      console.log(2);
      setTimeout(() => {
        console.log(3);
      }, 500);
    }, 2000);
  }, 1000);
}
```

Unübersichtlicher Code

- In dem Code ist ggf. nicht einfach erkennbar, welche Teile
  - den ausführenden Code
  - den Callback-Anteil im Erfolgsfall
  - den Callback-Anteil im Fehlerfall
- darstellen
- Insbesondere da oft anonyme Funktionen oder Lambda Ausdrücke verwendet werden
- Wird eine Aufgabe fortgesetzt, besteht auch diese wieder aus drei Teilen
- Auch eine Fortsetzung kann fortgesetzt werden, usw.

Was sind Promises?

- Promise => Versprechen
  - Werden auch Deferreds oder Futures genannt
- Repräsentieren das mögliche Ergebnis einer asynchronen Operation
- Können einer Variablen zugewiesen werden
  - oder
- als Parameter einer Funktion übergeben werden

Was sind Promises?

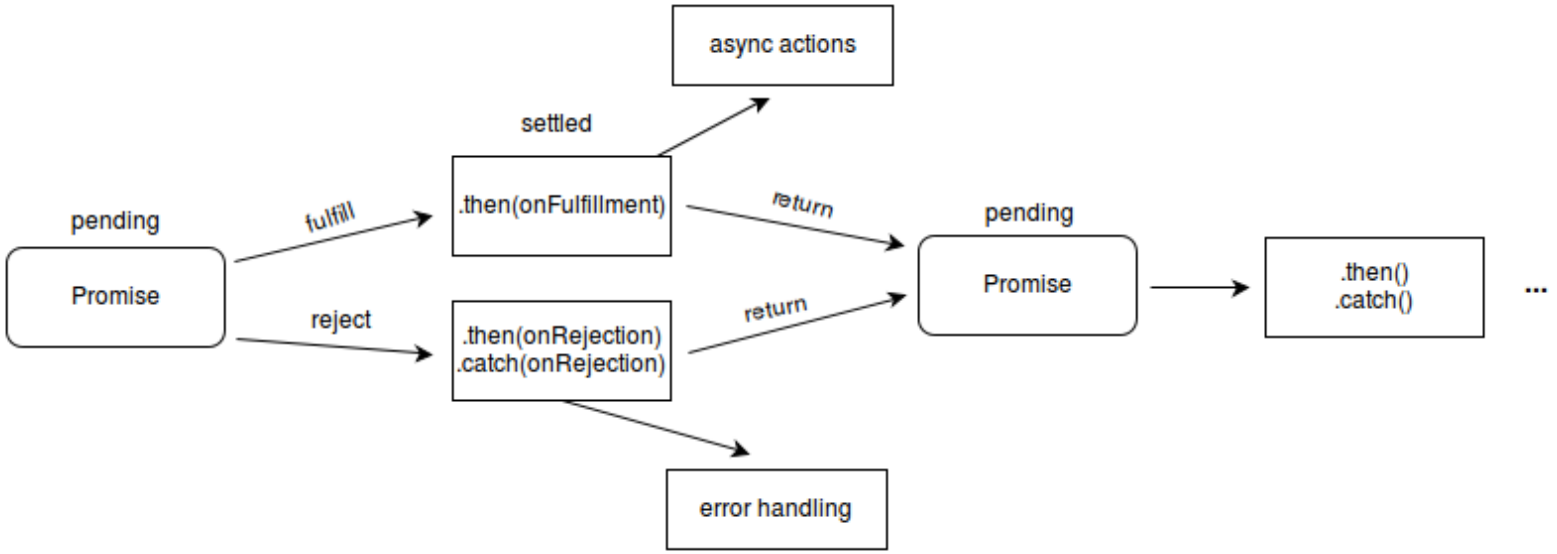
- Eine Funktion liefert ein Versprechen
- Wird das Versprechen eingehalten, kann die nächste Aufgabe ausgeführt werden und das Versprechen wird an diese übermittelt
- Wird das Versprechen nicht eingehalten (z.B. bei einem Fehler), wird der Fehler an einer entsprechenden Stelle abgefangen
- Das Promise wird asynchron ausgeführt und blockiert daher nicht den Programmablauf

Promise

- Promises haben gewisse Ähnlichkeiten zu Event-Listenern
- Ein Promise kann jedoch nur einmal erfolgreich sein oder scheitern
- Es kann z.B. nicht zweimal erfolgreich sein oder zweimal fehlschlagen
- Es kann auch nicht von Erfolg zu Misserfolg oder umgekehrt wechseln

Promise

- Ein Promise kann
  - `pending` — laufend
  - `fullfilled` — erfolgt
  - `rejected` — abgelehnt
  - `settled` — erfolgt oder abgelehnt
- sein



Promise verwenden

- Um ein Promise zu erzeugen, wird dem Konstruktor eine Funktion oder Lambda-Ausdruck als Parameter übergeben

- Diese Funktion ist der so genannte *Exekutor*
  - Im Exekutor wird etwas ausgeführt, was lange dauern kann

- Der Exekutor wird sofort aufgerufen und bekommt zwei Callbackfunktionen als Parameter übergeben
  1. den Resolver (bei Erfolg)
  2. den Rejector (im Fehlerfall)

- `new Promise(function(resolve, reject) { ... });`

Zustände

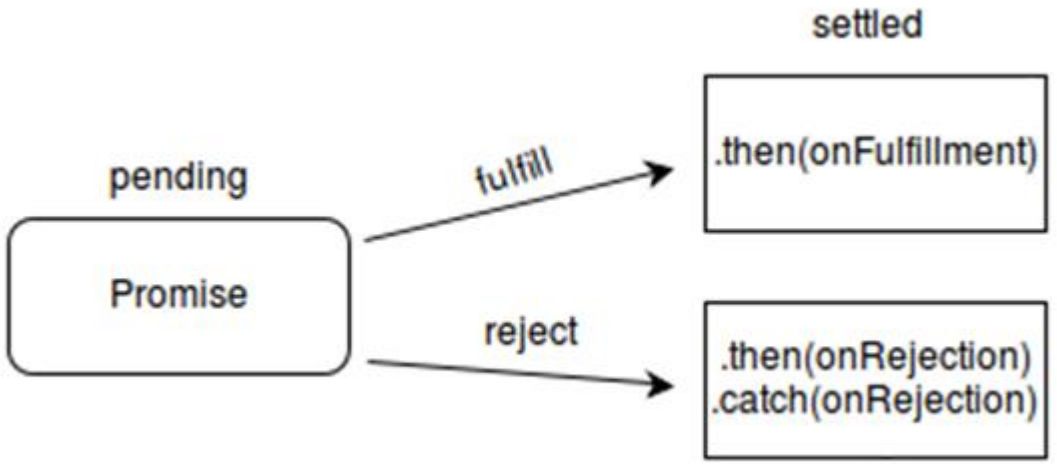
› Solange der Exekutor nicht den Resolver oder den Rejector aufgerufen hat, ist das Promise in einem Schwebezustand (`pending`)

› Nachdem einer der beiden aufgerufen wurde, ist das Promise festgelegt (`settled`) und zwar entweder auf

  erfüllt (`fulfilled`)

› oder

  zurückgewiesen (`rejected`)



Zustände - Erfolg

- Der erste Parameter ist die Resolver-Funktion (kann anderen Namen verwenden)
- Der Funktion können Parameter übergeben werden
  - Diese dienen als Rückgabewert bzw. Ergebnis des Promise
- Wird die Funktion im Laufe der Ausführung aufgerufen, wurde das Promise erfolgreich ausgeführt (Promise fulfilled)

```
let promise = new Promise((resolve, reject) =>
{
 resolve('Fehlerfrei!');
});
```

```
let promise = new Promise((resolve, reject) => {
  resolve('Fehlerfrei!');
})
undefined
promise
▼ Promise {<fulfilled>: 'Fehlerfrei!'} i
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: "Fehlerfrei!"
```

Zustände - Fehlerfall

› Der zweite Parameter ist die Rejector-Funktion

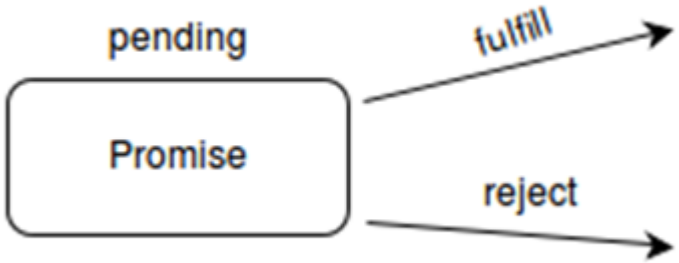
› Wird die Funktion im Laufe der Ausführung aufgerufen, wurde das Promise nicht erfolgreich ausgeführt (Promise rejected)

```
let promise = new Promise((resolve, reject) =>
{
 reject('Problem!');
});
```

```
let promise = new Promise((resolve, reject) => {
 reject('Problem!');
});
undefined
▶Uncaught (in promise) Problem!
promise
▼Promise {<rejected>: 'Problem!'} i
  ▶[[Prototype]]: Promise
   [[PromiseState]]: "rejected"
   [[PromiseResult]]: "Problem!"
```

Ein Promise erzeugen

```
let promise = new Promise(function(resolve, reject) {
  // etwas (asynchron) tun
  if (/* alles geklappt hat */)
  {
    resolve("Alles gut!");
  }
  else
  {
    reject("Fehlerfall :( ");
  }
});
```



Struktur durch Promises

- Beim Lesen des Codes eines Promises wird deutlich
- Was ausgeführt werden soll (Exekutor)
- Was im Erfolgsfall danach ausgeführt werden soll (Resolver)
- Was im Fehlerfall danach ausgeführt werden soll (Rejector)

- Diese Struktur stellt damit den wesentlichen Unterschied zur reinen Verwendung von Callbacks dar

- Für die Fortsetzung von Promises gibt es ebenfalls Methoden, die den Zweck deutlich machen

Ein Promise konsumieren

- Ein Promise liefert Ergebnisse in der Zukunft zurück, auf welche später zugegriffen werden soll

- Promises haben eine Methode namens **`then`** , die dann aufgerufen wird, wenn ein Promise den Resolver aufgerufen hat

- `then` liefert den Wert des Promises als Übergabeparameter zurück

Ein Promise konsumieren

```
let promise = new Promise((resolve, reject) =>
{
  resolve('Fehlerfrei!');
});

promise.then(ergebnis =>
{
  console.log(ergebnis);    // Ausgabe: Fehlerfrei!
});
```

Ein Promise konsumieren

```
console.log("1");
let promise = new Promise((resolve, reject) => {
  console.log("2");
  // Den Resolver erst nach 2 Sekunden aufrufen
  setTimeout(() => resolve('Resolved!'), 2000);
});

console.log("3");

promise.then(response => {
  console.log("5");
  console.log(response);
});
console.log("4");
```

Ausgabe:
1
2
3
4
5
Resolved!

Promise als Rückgabewert

```
function myAsyncFunction(a, b)
{
    return new Promise(function (resolve, reject) {
        // etwas spannendes tun...
        if ( a > b ) {
            resolve("Läuft");
        }
        else {
            reject("Kaputt");
        }
    });
}

› myAsyncFunction(100, 5).then(ergebnis => console.log(ergebnis));
```

Verkettung

› Promises können verkettet werden

› Das erste `then` kann Daten für den zweiten Aufruf von `then` anhängen

```
promise.then(wert1 => wert1 + ' Extra Daten!' )
       .then(wert2 => console.log(wert2));
```

Auf mehrere Promises warten

- Die `Promise.all()`-Methode nimmt ein Array mit Promises als Eingabe entgegen und gibt ein einzelnes Promise zurück
- Dieses zurückgegebene Promise wird erfüllt, wenn alle Promises der Eingabe erfüllt werden
  - Liefert ein Array der Erfüllungswerte
- Es lehnt mit dem ersten Ablehnungsgrund ab, wenn eines der Versprechen abgelehnt wird

- Es gibt zudem noch `Promise.any()`

Fehlerfall

› Der Fehlerfall kann mittels `catch` abgefangen werden

```
let promise = new Promise((resolve, reject) =>
{
 reject('Problem!');
});

promise.catch(fehlertext => console.log(fehlertext));
```

Fehlerfall

```
let promise = new Promise((resolve, reject) =>
{
 reject(Error('Problem!'));
});
```

› Oft ist es sinnvoll ein mit einem Error-Objekt abzulehnen

› Fehlerobjekte erfassen einen Stack-Trace, wodurch Debugging-Tools hilfreicher werden

Erfolgs- und Fehlerfall - Alternativen

```
let promise = new
Promise(function(resolve,
reject) { … });
// then & catch
promise.then(response =>
 console.log(response));

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

```
let promise = new
Promise(function(resolve,
reject) { … });
// then mit 2 Funktionen
// 1. resolved, 2. rejected
promise.then(
response =>
console.log(response)
,
 error => console.log(error)
);
```

finally

- So wie es bei `try/catch` eine `finally`-Klausel gibt, existiert `finally` auch bei Promises
- `finally` wird immer ausgeführt, egal ob `resolved` oder `rejected`, d.h. sobald das Promise `settled` ist
  - Der Aufruf von `finally` ist ähnlich zu dem Aufruf von `then` mit zwei Funktionen
- `finally` ist daher ideal für Aufräumarbeiten
  - z.B. stoppen von Ladebalken, etc.