Kapitel 6.60 - Promises (/buch#angularjs-promises)

Was sind Promises und welche Probleme lösen sie?

Um Promises verständlich zu machen, fangen wir mit einer groben Umschreibung an und gehen dann auf Details und konkrete Anwendung ein. Wenn ihr euch erstmal unter dem Begriff **Promise** nichts vorstellen könnt, seid ihr nicht allein. Promises sind so etwas wie Callbacks 2.0. Wir möchten auch mit Promises asynchrone Probleme lösen. Hier ein Vergleich:

Beispiel Promise:

```
$http.get('/meine/api.json').then(
  function(response){ ... },
  function(error){ ... }
);
```

Beispiel Callback:

```
$.get('/meine/api.json')
  .done(function(){ ... })
  .fail(function(){ ... });
```

Auf den ersten Blick ist dabei kein großer Unterschied zu sehen. Da Promises in vielen Fällen ähnlich zu Callbacks aussehen, beantworten wir erstmal die Frage, was uns bei normalen Callbacks fehlen könnte:

Übersichtlichkeit

Problem:

Wenn wir mehrere asynchrone Aufrufe mit Callbacks verschachteln, wird es sehr schnell unübersichtlich. Dieses Probleme der Übersichtlichkeit ist so massiv, dass es sogar einen eigenen Namen bekommen hat: **Pyramid of Doom**. Hier ein kleines Beispiel:

```
$.get('/api1').done(function(data) {
  $.get('/api2').done(function(data) {
    $.get('/api3').done(function(data) {
    });
  });
});
```

Lösung:

Ein Promise gibt wieder ein Promise zurück. Somit können wir beliebig Promises beliebig verketten. Nehmen wir als Beispiel folgendes Promise:

```
var neuesPromise = $http.get('/meineApi').then(function(response) { retur
```

Das Ergebnis von `$http` ist wieder ein neues Promise. Dieses Promise enthält die Werte von `response.data`.

Als Rückgabewert können wir alle möglichen Datenstrukturen von JavaScript + Promises benutzen. So sehe eine Verkettung von Promises aus.

```
$http.get('/meineApi')
  .then(function(response) { return transform1(response) })
  .then(function(response) { return transform2(response) })
  .then(function(response) { return transform3(response) });
```

Synchronisation von mehreren asynchronen Aufrufen

Problem:

Wir haben z.B. mehrere APIs, die wir gleichzeitig abfragen möchten. Die Ergebnisse der APIs kommen in beliebiger Reihenfolge zurück. Mehrere Callbacks zu synchronisieren ist sehr aufwändig. Beispiel:

```
$.get('/api1').done(function(data){ result1 = data; });
$.get('/api2').done(function(data){ result2 = data; });
$.get('/api3').done(function(data){ result3 = data; });
```

Wie können wir jetzt an dieser Stelle auf feststellen, dass alle 3 APIs ihre Daten erfolgreich zurückgeliefert haben? - Wir müssten in jedem Callback überprüfen, ob die anderen APIs schon fertig sind. Damit wir das prüfen können, müssen wir noch ein zusätzliches Array erstellen, wo wir die Status der APIs zwischenspeichern. Möglich, aber nicht besonders elegant für ein Standard-Problem. Vor allen Dingen, wenn es eine gute Abstraktion dafür gibt.

Lösung:

Wie wir oben gesehen haben, ist das Synchronisieren von Callbacks nicht besonders elegant. Da dieser Fall aber recht oft auftritt, gibt es in der Promise-Bibliothek eine Lösung dafür: `$q.all()`.

$q.all nimmt ein Array von Promises entgegen und gibt selber wieder ein Promise zurück. Beispiel:

```
var api1 = $http.get('/api1);
var api2 = $http.get('/api2);
var api3 = $http.get('/api3);

$q.all([api1, api2, api3])
  .then(function(responsesArray) {
    $scope.resultAll = 'Resolved with ' + responsesArray;
  });
```

Der `.then()`-Block wird ausgeführt nachdem alle Promises erfolgreich aufgelöst wurden. Bekommen wir alle Ergebnisse erfolgreich zurück, befinden sich in `responsesArray` die Rückgabewerte der einzelnen Promises. Diese sind in der Reihenfolge, wie wir sie angegeben hatten. Sollte allerdings ein Aufruf fehlschlagen, wird direkt die Error-Funktion des Promises aufgerufen.

Fehlerbehandlung

Problem:

Bei verschachtelten Callbacks ist nicht definiert, wie wir mit Fehlern umgehen. Beispiel: Wir rufen die 3 verschachtelten Callbacks aus dem letzten Beispiel auf. Was passiert, wenn `/api3` einen Fehler wirft? Müssen wir eine Error-Behandlung für jeden Aufruf duplizieren?

```
$.get('/api1').done(function(data) {
  $.get('/api2').done(function(data) {
    $.get('/api3').done(function(data) {
    }).fail(function(){ ... });
  }).fail(function(){ ... });
}).fail(function(){ ... });
```

Schön wäre es, wenn wir die Fehlerbehandlung an einer Stelle lösen könnten.

Lösung

Value & Error Downstream Propagation

Vermischung von Verantwortlichkeiten

Problem:

In der Informatik gibt es das Prinzip der *Aufteilung nach Verantwortlichkeiten* (Separation of concerns (http://en.wikipedia.org/wiki/Separation_of_concerns)). Nehmen wir einen Standard-Callback als Beispiel:

```
$.get('/meine/api.json', function(data) {
  updateView();
  processData();
});
```

Wenn die unsere API abgerufen wurde, möchten wir etwas in der View aktualisieren mit `updateView()` und gleichzeitig die Daten weiterverarbeiten, mit `processData()`. Wir können uns sicher darauf einigen, dass das 2 sehr verschiedene Aufgaben sind.

Lösung:

TODO

Der Aufbau von Promises

An dieser Stelle kommen wir endlich zu Promises selbst. Wir zeigen, wie ein Promise aufgebaut ist und gehen dass auf die Lösung der oben genannten Probleme ein.

Ein Promise hat immer die gleiche Schnittstelle:

```
meinPromise.then(successFn, errorFn)
```

Ein Promise gibt einen Funktion `then()` mit 2 Argumenten zurück. Das Erste ist für den Erfolgsfall, das Zweite für den Fehlerfall. Was dabei wichtig ist: Das Ergebnis einen Promises ist ein Promise. Somit können wir auch solche Konstrukte bauen:

```
meinPromise.then(successFn, errorFn).then(successFn, errorFn)
```

Vollständiger Source:

```
<html ng-app>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.0.6/an
    <script src="application.js"></script>
  </head>
  <body ng-controller="MainController">
    <fieldset>
      <legend>Promise 1</legend>
      <button ng-click="resolve1()">Resolve</button>
      <button ng-click="reject1()">Reject</button>
      {{result1}}
    </fieldset>
    <fieldset>
      <legend>Promise 2</legend>
      <button ng-click="resolve2()">Resolve</button>
      <button ng-click="reject2()">Reject</button>
      {{result2}}
    </fieldset>
    <fieldset>
      <legend>Promise All</legend>
      {{resultAll}}
    </fieldset>
  </body>
</html>
```

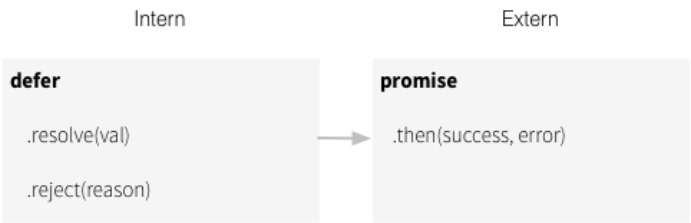
```
function MainController($scope, $q) {
  var defer1 = $q.defer();
  var defer2 = $q.defer();

  var promise1 = defer1.promise;
  var promise2 = defer2.promise;

  promise1.then(function(message) {
    $scope.result1 = 'Resolved with ' + message;
  }, function(reason){
    $scope.result1 = 'Rejected with ' + reason;
  });

  promise2.then(function(message) {
    $scope.result2 = 'Resolved with ' + message;
  }, function(reason){
    $scope.result2 = 'Rejected with ' + reason;
  });

  $q.all([promise1,promise2]).then(function(messages) {
    $scope.resultAll = 'Resolved with ' + messages;
  }, function(reason){
    $scope.resultAll = 'Rejected with ' + reason;
  });

  $scope.resolve1 = function() { defer1.resolve("Success1"); }
  $scope.reject1  = function() { defer1.reject("Error1");  }
  $scope.resolve2 = function() { defer2.resolve("Success2"); }
  $scope.reject2  = function() { defer2.reject("Error2");  }

}
```

Promises selbst schreiben

Ein Promise wird erzeugt mit der Funktion `$q.defer()`. Die defer()-Funktion gibt das Promise zurück.

```
var deferred = $q.defer();
return deferred.promise
```

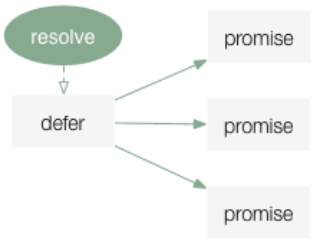
Es Promises haben also einen internen Teil (deferred) und einen externen Teil (promise):



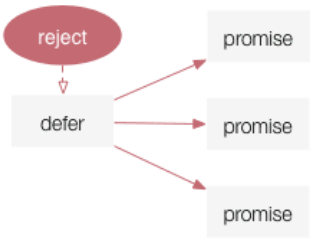
TODO

Nehmen wir als Beispiel ein kleines Würfelspiel. Wer eine 6 würfelt, gewinnt. Gewürfelt wir mit zwei Sekunden Verzögerung.

```
var wuerfel = $q.defer();
setTimeout(function() {
  ergebnis = Random...
  if (ergebnis == 6) {
    wuerfel.resolve(ergebnis);
  } else {
    wuerfel.reject(ergebnis);
  }
}, 2000);
promise = wuerfel.promise

promise.then(function(zahl) {
  alert('Erfolg: Du hast eine ' + zahl + ' gewürfelt!');
}, function(zahl) {
  alert('Fehler: Eine ' + zahl + ' reicht leider nicht');
});
```



promise.then(callback, errback)



promise.then(callback, errback)

« Internet Explorer - Kompatibilität (/buch/internet-explorer-kompatibilitaet) | Übersetzung/Internationalisierung (I18n) » (/buch/internation

von Sascha Brink (/entwickler/saschabrink)