	Raceconditions bei MySQL Abfragen		AnPr	V 1.2
	Name	Klasse	Datum	

1 Problemstellung

Sehen wir uns hierzu folgendes Problem an: wir wollen mit einer Anfrage an den Server eine gewisse Teildatenmenge – sagen wir die Datensätze 30 bis 35 lesen. Zusätzlich wollen wir die Info über die Gesamtzahl haben:

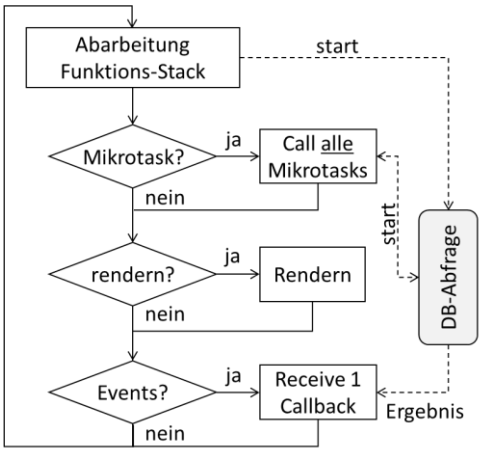
```
30 Eric Burdon
31 Etta James
32 Frankie Valli
33 Freddie Mercury
34 George Jones
Position 30 to 35 out of 100
```

Hierzu benötigen wir zwei SQL-Statements; eines für die Gesamtanzahl und eines für die eigentlichen Daten:

```
Musician.getCompleteList = (startPosition, resultData) => {
  const NO_OF_DISPLAY = 5; // Anzahl angezeigter Datensätze
  let noOfDataSets =  0;
  dbcon.query('SELECT COUNT(*) AS nubOfMusicians FROM musician',
  (err, result) => {
    if (err) {
      resultData(err, null);
      return;
    }
    noOfDataSets = result[0].nubOfMusicians;
  });
  // Sicherstellung, dass startPosition garantiert kleiner Anzahl ist
  if (noOfDataSets == 0) {
    startPosition = 0; // sollte noOfDataSets = 0 sein
  } else if (startPosition >= noOfDataSets && startPosition != 0) {
    startPosition = noOfDataSets - NO_OF_DISPLAY;
  }
  // Vorbereitung der SQL SELECT Limitierung
  const SQL_LIMIT = ` LIMIT ${startPosition}, ${NO_OF_DISPLAY}`;
  dbcon.query('SELECT ID AS musId, FName AS firstName, LName AS lastName
  FROM musician' + SQL_LIMIT, (err, result) => {
    if (err) {
      resultData(err, null);
      return;
    }
    let musicians = Array();
    if (result.length) {
      result.forEach((element) => {
          musicians.push(new Musician({...element}));
      });
      resultData(null, musicians, {start: startPosition,
      noData: noOfDataSets, noDisp: NO_OF_DISPLAY});
      return;
    }
    resultData({ kind: "not_found" }, null);
  });
};
```

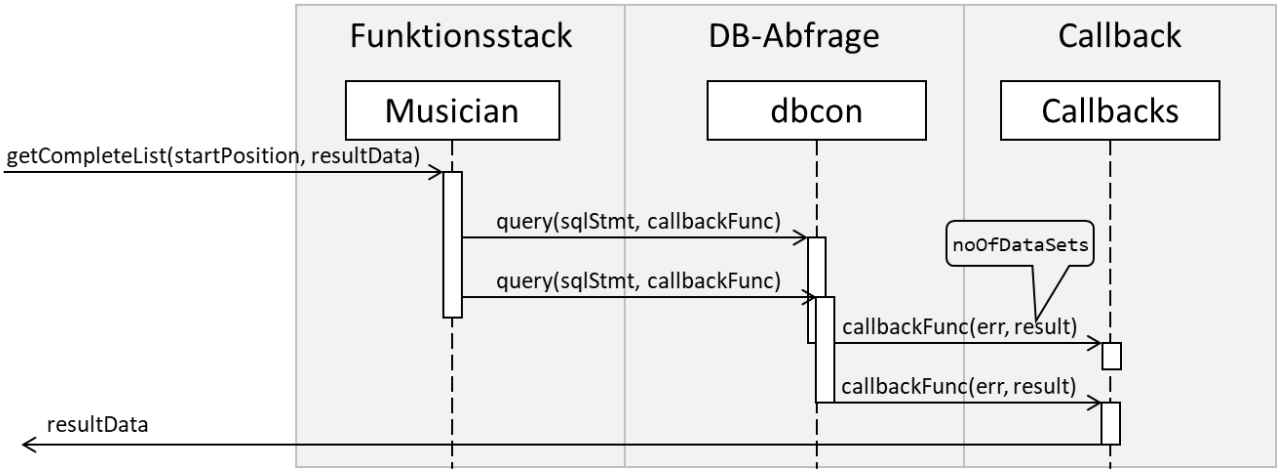
Wir rufen die Funktion mit einer eigenen Route `list/: startPosition` auf. Lassen wir den Code laufen, so erhalten wir immer die ersten 5 Positionen, egal welchen Startpunkt wir haben, Dies ist immer dann der Fall, wenn startPosition auf 0 gesetzt wird – **noOfDataSets** scheint immer 0 zu sein. Diese Fehlfunktion müssen wir nun analysieren.

2 Die Event Loop von JavaScript

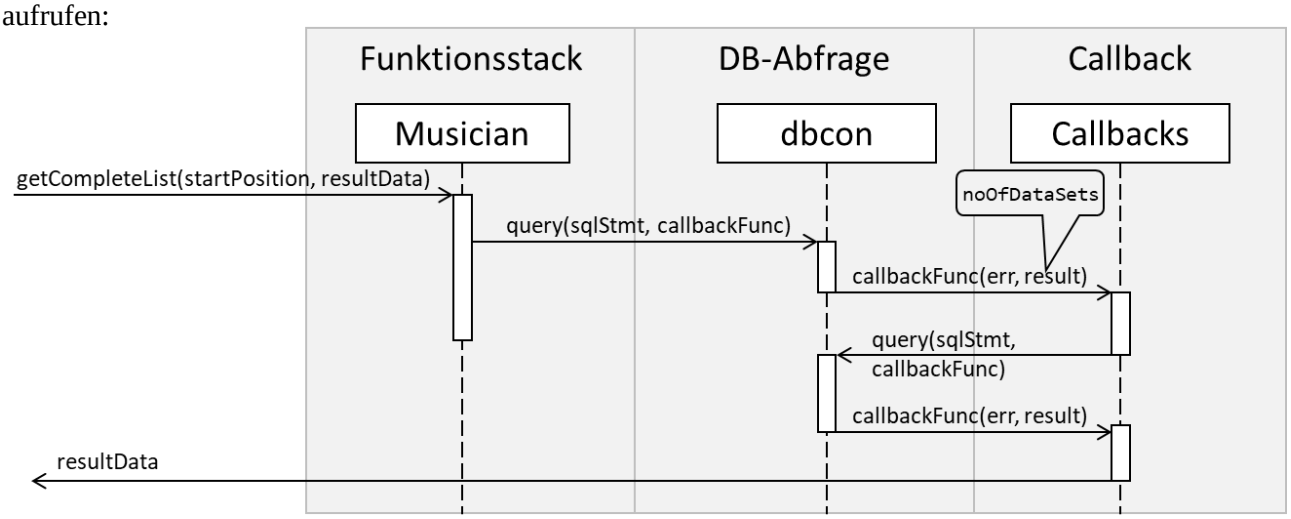


Die JavaScript Engine arbeitet den Code immer sequenziell ab – es gibt keine Parallelisierung! Rechts ist eine vereinfachte Darstellung der Event-Loop in JavaScript. Der „Funktions-Stack“ ist hierbei unser „normaler“ Code, der verarbeitet wird. In unserem obenstehenden Codebeispiel ist es alles, was nicht kursiv gedruckt ist. Nach der Verarbeitung wird im Browser gerendert bzw. in node.js die Ausgabe durchgeführt.

Gibt es nun Code, der auf eine Rückmeldung warten muss – so wie bei einer Datenbankabfrage – würde die Verarbeitung einfach pausieren. Um dieses „Einfrieren“ von Applikationen zu verhindern, gibt es nun die Möglichkeit, Elemente in der Verarbeitung auszulagern. Eine davon ist es, mit „Callbacks“ zu arbeiten. Hierbei wird wird die eigentliche Datenbankabfrage (bei uns ist es der `dbcon.query` Aufruf) außerhalb des „normalen“ JavaScript-Codes ausgelagert und abgearbeitet. Sobald ein Ergebnis vorliegt, platziert die Abfrage einen Callback-Event im Callback Stack, welches nach dem Rendern abgearbeitet wird.



Wie wir sehen, werden die beiden `query` Aufrufe sofort hintereinander aufgerufen (und quasi-parallel abgearbeitet), ohne dass die 2. `query` auf die 1. wartet. Der Wert `noOfDataSets` als Ergebnis der 1. `query` kann also noch nicht als Eingangsinformation der 2. `query` verwendet werden – er ist also noch auf dem Wert 0. Wenn wir also das Ergebnis der 1. Query für die 2. Query verwenden wollen, muss die 1. Query die 2. Query aufrufen:



Diese Verschachtelung von Callbacks macht den Code nicht unbedingt übersichtlicher und wird mitunter auch als „Callback Hell“ bezeichnet. Eine andere – übersichtlichere – Möglichkeit dieses Problem zu lösen ist es, mit Hilfe eines Promises einen „Microtask“ einzustellen, der vor dem Rendern abgearbeitet wird. Neben dem Verarbeitungszeitpunkt ist der Unterschied zwischen den Microtasks und den Callbacks, dass in einem Zyklus alle Tasks in einem Durchgang abgearbeitet werden, vom Callback Stack wird jedoch nur ein Event verarbeitet.

3 Callback-Test

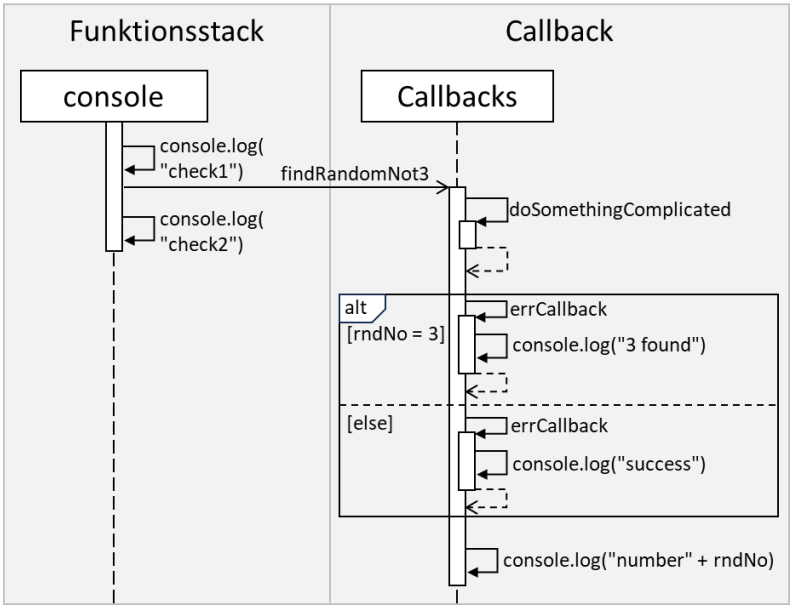
Sehen wir uns folgenden „Simulationscode“ an, welcher das Verhalten einer Anfrage mit möglichem Fehler abbildet. Das „externe“ Abarbeiten realisieren wir über „`setTimeout()`“ und einer Wartezeit von 0. Dadurch haben wir die Möglichkeit, Code aus der Sequenz des Funktionsstacks in den Callback-Stack auszulagern.

```
console.log("check1");
setTimeout(() => {
findRandomNot3(10, (succMessage, randNo) => {
        console.log(succMessage + "\nNumber: " + randNo);},
                    (errMessage) => {
        console.log(errMessage);
    });
}, 0);
console.log("check2");

// fake Belastung simuliert warten
function doSomethingComplicated() {
    let x = 0;
    for (let i = 0; i < 1000000000; i++) { x += Math.sqrt(i); }
    return x;
}

// fake Funktion, welche bei Zufallszahl = 3 einen Fehler wirft
function findRandomNot3(maxNo, succCallback, errCallback) {
    doSomethingComplicated();
    let rndNo = Math.floor(Math.random() * maxNo);
    if (rndNo != 3) {
        succCallback("success", rndNo);
    } else { errCallback("3 found"); }
}
```

Sehen wir uns erstmal die Aufrufstruktur an. Der normale Funktionsstack arbeitet den Code von „check 1“ bis „check 2 ab“. Dazwischen wird `findRandomNot3()` aufgerufen – allerdings über setTimeout, weshalb diese Funktion außerhalb des Funktions-Stacks ausgeführt wird. Die Funktion erzeugt eine Zufallszahl innerhalb einer vorgegebenen Obergrenze. Sollte die Zahl eine 3 sein, so wird dies als „Fehler“ interpretiert – so simulieren wir einen Fehlerfall.



Die Funktion erwartet drei Parameter – einmal eine Zufallsobergrenze (in unserem Fall 10) und eine Callbackfunktion für den Erfolgs- und eine für den Fehlerfall. Diese übergeben wir als anonyme

Funktionen, welche einfach nur Ausgaben loggen. Im Erfolgsfall wird eine Nachricht („succMessage“) und eine Zahl („randNo“) übergeben. Im Fehlerfall nur eine „errMessage“.

Innerhalb `findRandomNot3()` heißen die beiden Funktionen `succCallback` und `errCallback`. Bevor die Zufallszahl ermittelt wird, rufen wir jedoch noch eine weitere Funktion auf, welche einfach nur CPU-Last erzeugt, indem die Wurzel aus allen Zahlen zwischen 0 und 1.000.000.000 aufsummiert und zurückgegeben wird. Danach wird die Zufallszahl ermittelt und im Fall von 3 `errCallback` aufgerufen, ansonsten `succCallback`. Führen wir den Code aus, sehen wir unsere Aufrufstruktur bestätigt (hier mit 2 als Zufallszahl):

```
check1
check2
success
Number: 2
```

Die Erfolgsmeldung kommt also nach `check2`! Der „normale“ Code muss also im Funktions-Stack nicht auf die aufwändige Wurzelberechnung warten. Die Wartezeit ist zwischen check2 und success.

4 Promises

Ein Promise löst das Problem nun anders, und zwar mit Hilfe von Microtasks. Sehen wir uns hierfür den Code an:

```
console.log("check1");

function findRandomNot3(maxNo){
  return new Promise((resolve, reject) => {
    doSomethingComplicated();
    let rndNo = Math.floor(Math.random() * maxNo);
    if (rndNo != 3) {
      resolve({resultStatus: "success", rndNumber: rndNo});
    } else {
      reject({error_reason: "3 found"});
    }
  });
};

findRandomNot3(10).then((messgObj) => {
  console.log(messgObj.resultStatus + "\nNumber: " +
  messgObj.rndNumber);
}).((messgObj) => {
  console.log(messgObj.error_reason);
});
console.log("check2");
```

`findRandomNot3()` wird im Rahmen eines `Promise` – also als Microtask – ausgeführt, und somit auch außerhalb des Funktions-Stacks. Dies wird dadurch erreicht, dass unsere Funktion ein Promise Objekt zurückgibt. Promises benötigen immer eine Erfolgs- und eine Fehlermeldung (`resolve` und `reject`). In unserem Fall ist der Fehler die Zufallszahl 3. Wenn nun `findRandomNot3()` ausgeführt wird, gibt er das Promise Objekt zurück und wir können die Methoden des Promise aufrufen. Ein `Promise` erlaubt im Erfolgsfall den „`then()`“ Aufruf, ansonsten den „`catch()`“ Call. Führen wir den Code aus, sehen wir wieder `check1` und `check2` vor der `success` Meldung:

```
check1
check2
success
Number: 2
```

Allerdings ist die Wartezeit woanders, als beim Callback. Zuerst kommt `check1`, danach die Pause und dann den gesamten Rest, da immer alle Microtasks abgearbeitet werden müssen und das rendering nach den Microtasks erfolgt. Wenn wir den Code oft genug ausführen, werden wir auch einen Fehlerfall sehen (also die 3).

5 Sequenz von Promises

Ergänzen wir unser kleines Testprogramm um eine weitere Funktion, welche ebenfalls eine Zufallszahl generiert und um die erste Zufallszahl erhöht – wir haben also eine Abhängigkeit der 2. Funktion zur 1. Funktion (ganz ähnlich wie bei unseren beiden SQL-Abfragen vom Anfang). Hier soll der Fehlerfall die 4 sein. Wenn wir diese Funktion ebenfalls als Promise realisieren, können wir die beiden Promises kaskadieren, so dass der eine auf den anderen wartet:

```
console.log("check 1");
findRandomNot3(10).then((messgObj) {
  console.log(messgObj.resultStatus + "\nNumber: " + messgObj.rndNumber);
  return addRandomNot4(10, messgObj.rndNumber);
}).then((messgObj) => {
  console.log(messgObj.resultStatus + "\nSum: " + messgObj.addNumber);
}).catch((messgObj) => {
  console.log(messgObj.error_reason);
});
console.log("check 2");
…
function addRandomNot4(maxNo, addNo) => {
  return new Promise((resolve, reject) => {
    doSomethingComplicated();
    let rndNo = Math.floor(Math.random() * maxNo);
    console.log("add number: " + rndNo);
    if (rndNo != 4) {
      resolve({resultStatus: "success", addNumber: rndNo + addNo});
    } else {
      reject({error_reason: "4 found"});
    }
  });
}
```

Wir können also ein „`then`“ an das andere hängen – den „`catch`“ – Block teilen sich beide Aufrufe.

6 async / await

Nun ist die Verkettung von mehreren Promises zwar relativ einfach zu bewerkstelligen, jedoch sieht der Code nicht sehr „schön“ aus, da er in einem einzigen Konstrukt „zusammengeklebt“ wirkt. Hier kommt nun `async` / `await` ins Spiel, was lediglich ein anderer Syntax ist, das oben gezeigte Konstrukt zu realisieren. Man spricht hier von „syntaktischem Zucker“:

```
console.log("check 1");
calculateRandoms();
console.log("check 2");
async function calculateRandoms() {
  try {
    const n3Res = await findRandomNot3(maxNo);
    console.log(n3Res.resultStatus + "\nNumber: " + n3Res.rndNumber);
    const n4Res = await addRandomNot4(10, n3Res.rndNumber);
    console.log(n4Res.resultStatus + "\nSum: " + n4Res.addNumber);
    } catch(messgObj) {
        console.log(messgObj.error_reason);
    };
}
…
```

Entscheidend sind nun die Schlüsselwörter `async` und `await`. `async` wird benötigt, um dem Interpreter mitzuteilen, dass die gesamte Funktion als Microtask auszuführen ist, um ggf. warten zu können. Der Interpreter darf also „aus der Funktion springen“, wenn es notwendig ist. Die Notwendigkeit zu warten, wird über `await` indiziert.

Mit diesem Wissen können wir nun unser Problem vom Anfang fixen. Erzeugen wir erstmal eine Funktion, welche die SQL-Query in einen Promise packt:

```
function sqlQuery(sqlQuery, params, dbcon) {
  return new Promise((resolve, reject) => {
    dbcon.query(sqlQuery, params, (err, result) => {
      if (err) {
        reject(err);
      } else {
        resolve(result);
      }
    });
  });
};
```

Dies kann ggf. auch in ein eigenes JavaScript File ausgelagert werden (`module.exports = sqlQuery;` hier nicht vergessen!). Nun wenden wir diese Funktion an, um die Datenbankkommunikation als `async` zu definieren:

```
Musician.getCompleteListAsAw = async (startPosition, resultData) => {
  const NO_OF_DISPLAY = 5;

  try {
    const noOfDataSetsResult = await sqlQuery(
    'SELECT COUNT(*) AS nubOfMusicians FROM musician', [], dbcon);
    const noOfDataSets = parseInt(noOfDataSetsResult[0].nubOfMusicians);

    if (noOfDataSets == 0) {
      startPosition = 0;
    } else if (startPosition >= noOfDataSets && startPosition != 0) {
      startPosition = noOfDataSets - NO_OF_DISPLAY;
    }
    const SQL_LIMIT = ` LIMIT ${startPosition}, ${NO_OF_DISPLAY}`;
    const dataSetsResult = await sqlQuery(
    'SELECT ID AS musId, FName AS firstName, LName AS lastName FROM
    musician' + SQL_LIMIT, [], dbcon);
    let musicians = Array();
    dataSetsResult.forEach((element) => {
      musicians.push(new Musician({...element}));
    });

    resultData(null, musicians, {start: startPosition,
    noData: noOfDataSets, noDisp: NO_OF_DISPLAY});
    return;
  } catch(err) {
    resultData(null, err, null);
  }
};
```

Führen wir den Code nun aus, sehen wir das erwartete Ergebnis.