WBE-Praktikum 6

Files und asynchrone Programme

Aufgabe 1: CSV zu JSON

Ziel dieser Aufgabe ist es, ein Script zu implementieren, mit dem Sie eine CSV-Datei (Comma-Separated Values) nach JSON konvertieren können:

```
$ node csv2json.js data.csv data.json
```

Implementieren Sie zunächst den Teil, welcher die CSV-Datei mit Hilfe der File System API synchron einliest und verschiedene Informationen zur Datei ausgibt:

- Grösse der Datei
- Datum der letzten Änderung
- Anzahl Datensätze
- Benötigte Zeit zum Lesen der Datei

Zum Testen sind unter *code/csv* eine CSV-Datei abgelegt.

Ergänzen Sie das Programm so, dass die eingelesenen Daten in JSON konvertiert und in die Ausgabedatei geschrieben werden. In JSON resultiert ein Array, welches für jede der CSV-Zeilen einen Eintrag enthält. Die Attribute befinden sich in der ersten Zeile der CSV-Datei. Sie werden in jeden Eintrag im JSON-Array als Attribute des Objekts übernommen.

Hinweise:

- Passen Sie auf, dass Sie durch falsche Interpretation der Kommandozeilenargumente nicht versehentlich Ihr Script mit der JSON-Datei überschreiben, wenn beides im gleichen Verzeichnis liegt.
- Die Methode *split* von *String.prototype* zerlegt einen String an vorgegebenen Trennzeichen in seine Bestandteile: https://devdocs.io/javascript/global_objects/string/split
- Sie können zunächst ein JavaScript-Objekt aufbauen, welches mit JSON.stringify in einen JSON-String konvertiert wird: https://devdocs.io/javascript/global_objects/json/stringify

Weitere Aufgaben:

- Vergleichen Sie die Zeit zum Lesen der Datei mit der für die Verarbeitung benötigten Zeit.
- Verwenden Sie zum Lesen und Schreiben der Dateien nun asynchrone Funktionen.
- Testen Sie auch die Promise-Varianten der Lese- und Schreibfunktionen.

Aufgabe 2: Asynchroner Code und Promises

Im Verzeichnis *code/async* finden Sie eine Reihe von Beispielscripts, welche nun für eine Reihe von Versuchen mit Callbacks und Promises verwendet werden sollen:

- Im Script *naive-async.js* soll eine Datei geschrieben und wieder eingelesen werden. So funktioniert es aber nicht. Korrigieren Sie den Fehler.

- Betrachten Sie das Script *immediate.js* und überlegen Sie erst, in welcher Reihenfolge die Ausgaben erfolgen werden, bevor Sie Ihre Annahme durch Ausführen des Scripts überprüfen. Bis auf die Ausgabe von „script started" erfolgen alle Ausgaben erst in der Event Loop. Dies gilt auch für ein *setTimeout* mit Verzögerung 0. Die Reihenfolge von „timeout" und „immediate" ist laut Spezifikation undefiniert, da ein Timeout in Node.js immer mit einer kleinen Verzögerung von 1ms versehen wird. Dagegen erscheint „immediate from readFile callback" immer vor „timeout from readFile callback". Warum?

- Erklären Sie auch die Reihenfolge der Ausgaben von *nexttick.js*.

- In *emitter.js* wird ein *EventEmitter* verwendet. Überlegen Sie, in welcher Reihenfolge die Ausgaben erfolgen, bevor Sie Ihre Annahmen verifizieren.

- Machen Sie ein paar Versuche mit den Scripts *promise-wait.js*, *promise-data.js*, *promise-chain.js*, *promise-race-all.js*, sowie *promise-demo.js*.

- Im Script *promise-priority.js* werden verschiedene Queues in der Event Loop mit Handlern gefüllt. In welcher Reihenfolge diese ausgeführt werden ist einigermassen vorhersehbar, wenn man die Abläufe in der Event Loop kennt.

- Das Script *async-await.js* demonstriert den Umgang mit *async* und *await*. In der Funktion *add* sind die beiden Variablen a und b Platzhalter für Werte, die erst zu einem späteren Zeitpunkt verfügbar sein werden. Probieren Sie es aus. Was geschieht, wenn die beiden *await* vom *return*-Ausdruck vor die *resolveAfter2Seconds*-Aufrufe verschoben werden? Zum Vergleich ist noch eine Variante mit *Promise.all* angegeben.

- Als kleine Vorschau auf Browsertechnologien: Betrachten Sie *load-event.html*. Nach dem Laden des Bilds und dem Anlegen eines Listeners wird das Script mit einer aktiven Schleife blockiert. Öffnen Sie das Beispiel im Browser. Erst nachdem die drei Sekunden verstrichen sind, können weitere Ereignisse verarbeitet werden. Auch das Bild wird erst dann angezeigt.

  Wichtig:
  Don't block the Event Loop.

Aufgabe 3: Module

In verschiedenen Beispielen – unter anderem mit Jasmine – haben wir bereits das Modulsystem von Node.js (CommonJS) verwendet. Mittlerweile unterstützt Node.js auch ES6-Module, so dass nun auf der Browser-Plattform und unter Node.js ein einheitliches Modulsystem verwendet werden kann. Der Vollständigkeit halber soll auch das ES6-Modulsystem noch kurz ausprobiert werden.

Nehmen Sie eine der in den letzten Praktikumslektionen geschriebenen Funktionen, zum Beispiel die Funktion *findTag*, und erstellen Sie eine Kopie der betreffenden JavaScript-Datei. Ändern Sie die Erweiterung der Datei auf *.mjs* – damit wird für Node.js angegeben, dass das ES6-Modulsystem verwendet werden soll. Alternativ könnte man eine *package.json*-Datei mit folgendem Eintrag anlegen: `{ "type": "module" }`

Am Ende der Moduldatei muss die *findTag*-Funktion exportiert werden.

Erstellen Sie nun eine Programmdatei, zum Beispiel *prog.mjs*, in welcher die *findTag*-Funktion importiert und aufgerufen wird.