Asynchrone Programmierung

https://iuk.one/1066-1007

Clemens H. **Cap**
ORCID: 0000-0003-3958-6136

Department of Computer Science
University of **Rostock**
Rostock,Germany
clemens.cap@uni-rostock.de

Version 1

1. Beispiele zum Einstieg

Um welche Problematik geht es hier genau?

Synchrone Programmierung

```
var data;
data = getDataFromServer();  // blockiert
var displayData = workOnData (data);
view.insert (displayData);
```

Src. 1: Das Grundproblem der synchronen Programmierung sind blockierende Aufrufe: Während ein Aufruf auf dem Server blockiert, kann der User am Client nichts tun, weil der entsprechende Thread warten muß.

Frage: Wie löst man das Problem des Wartens auf blockierende Aufrufe?

Aufgabe: Wiederhole Problem blockierender Aufrufe!

Wiederhole: Wie geht Informatik mit dem Problem blockierender Aufrufe um?

Denke an:

- **Betriebssystem:** Ein langsames Gerät wird gelesen (Bsp: Festplatte).
- **Dialogsystem:** Benutzer bewegt die Maus während Anwendung arbeitet.
- **Netzwerk:** Mehrere Netzwerk-Karten sind gleichzeitig aktiv.

Probleme: Deadlock / Livelock / Fairness

Deadlocks:

- Bsp: Alice wartet auf Bob, der auf Carol wartet, die auf Alice wartet.
- Wir erinnern uns an die **4 Bedingungen von Coffman** für ein Deadlock: Mutual exclusion, no preemption, hold and wait, circular wait.
- Zufälligkeit des Task-Wechsels erzeugt Deadlock.
- Deadlock verzögert Prozeß dann unendlich lange.
- **Lösungen:** Coffman durchbrechen; deadlock detection; deadlock prevention.

Livelocks:

- Kein Fortschritt im Sinne der Spezifikation.
- Zufälligkeit des Task-Wechsels verzögert Prozeß endlich, randomisiert beliebig lange.
- **Problem:** Lösungsversuch lokaler Fairness reicht oft nicht aus.
- **Lösung:** Global faire Scheduler.

Probleme: Race Conditions

Auf Architekturen mit **einer** processing unit:

- Bsp: Eine CPU mit Thread/Prozeß Wechsel.
- Zufälligkeit des Task-Wechsels im Scheduler verändert Ergebnis.

Auf Architekturen mit **mehreren** processing units:

- Bsp: Hyperthreading und Multicore Architekturen.
- Unterschiede in Laufgeschwindigkeiten von Prozessen verändert Ergebnis.

Aufgabe: Finde Race Conditions (1)

```
function one () {
  local oldVal = readAccount();
  local newVal = oldVal - 30;
  writeAccount (newVal);
}

function two {
  local oldVal = readAccount();
  local newVal = oldVal - 20;
  writeAccount (newVal);
}
```

Src. 2: Alice und Bob haben als Paar ein gemeinsames Konto und gehen zufälligerweise genau gleichzeitig zum Geldautomaten mit den Transaktionen `one` und `two` laufen.

Aufgaben:

- Welche möglichen End-Kontostände sind (bei naiver Implementierung) möglich?
- Beschreiben Sie die Anforderungen an das erwünschte Programmverhalten!
- Wie kann dieses erreicht werden?

Aufgabe: Finde Race Conditions (2)

```
function one () {
  writeAccount (readAccount() - 30);
}

function two {
  writeAccount (readAccount() - 20);
}
```

Src. 3: Alice und Bob haben als Paar ein gemeinsames Konto und gehen zufälligerweise genau gleichzeitig zum Geldautomaten mit den Transaktionen `one` und `two` laufen.

Aufgaben:

- Ist das jetzt eine Lösung?
- Wenn ja: Warum?
- Wenn nein: Warum nicht?

Frage: Hier waren das Beispiele in Pseudocode. Wann können in Javascript solche Probleme auftreten?

Typische Hilfsmittel zur Synchronisation

Kategorie 1: Synchronisations-Primitiva

- Semaphore, Monitore, Condition Variables, Barrieren, Kritische Abschnitte, Mutexe, Locks uva.
- Basieren typischerweise auf Anhalte-Mechanismen (obstructions)

Kategorie 2: Obstruction-free algorithms.

- Neueres Forschungsgebiet.

Kategorie 3: Transaktions-Protokolle

- Siehe DB-Vorlesung

Verfügbarkeit in JavaScript:

- Browser: Mutex Library
- Node: Mutex Library
- Beide: async Library

Parallelitäts-Architektur im Browser

Browser ist **multi threaded** in der Implementierung.

- DNS requests.
- Parallele download requests.
- Paralleles parsing.
- Preloading.

Browser ist **single threaded** im Programmiermodell.

- Nur ein thread in der DOM.

Browser ist **multi threaded** in Erweiterungen.

- Web Workers, Service Workers
- Programmier-Modell: Message Passing.

2. Asynchrone Programmierung

Welche Techniken der asynchronen Programmierung gibt es in JS?

Welche weiteren Konzepte nicht-linearer Kontrollflüsse gibt es in JS?

Asynchrone Konzepte

Callbacks

Eine asynchrone Aktivität bekommt eine Funktion zugewiesen (sog. Callback), das nach Ende der Aktivität aufgerufen wird, sobald der Thread Zeit hat.

Events

Das Ende einer asynchronen Aktivität löst ein Ereignis aus. Sobald der Thread Zeit hat, wird der Handler des Ereignisses aufgerufen.

Promises

Synthetische Objekte, die als Platzhalter für später zu erhaltende Ergebnisse dienen.

Asynchrone Funktionen

Funktionen, die zu Beginn einer asynchronen Aktivität eine Unterbrechung erlauben.

Weitere Kontrollfluß-Konzepte

Ausnahmen: `try ... catch ... finally`

`setTimeout, setInterval`

`requestAnimationFrame` Automatisierte Zeittaktung

Web Worker: Kontrollierte shared-nothing Parallelität

- Wegen shared-nothing kaum Synchronisationsprobleme.
- Datenübergabe an message passing Schnittstellen.
- Echte Parallelität bei etwas größeren Tasks.

Service-Worker: Web Worker, die auch im Hintergrund laufen können.

Parallele Prozesse: Nur in node

Generatoren und `yield`

3. Callbacks

Wie funktionieren Callbacks?

Callback

```
function callback (data) {alert ("Antwort ist: " + data);}

getDataFromServer ( callback );
```

Src. 4: Programmierung mit Callbacks

Aspekte:

- Funktioniert bei synchronen und asynchronen Aufrufen.
- Typischerweise für asynchrone Aufrufe genutzt.

Problem: Lost in Continuation.

- Wo steht denn dann der Fortsetzungs-Code?

Callbacks bei Browser Geolocation

```
function success(pos) {
  console.log ("Position ist: ", pos);
}

function error (err) {
  console.log ("Geo Location Error: ", err);
}

if (!navigator.geolocation) {alert ( 'Geolocation nicht unterstützt!');}
else {navigator.geolocation.getCurrentPosition(success, error); }
```

Src. 5: Callbacks bei Browser Geolocation.

chmod in node (1)

```
import fs from 'fs';
fs.chmod('file.txt', 0o775, (err) => {
  if (err) throw err;
  console.log('Alles ok.');
});
```

Src. 6: chmod in node: Callback API.

```
import fs from 'fs';
try{
  fs.chmodSync ('file.txt', 0o775);
} catch (err) {console.error "Es gab einen Fehler: ", err);}
```

Src. 7: chmod in node: Synchrone API.

chmod in node (2)

```
import fs from 'fs';
fs.promises.chmod("example.txt", 0o400)
  .then  ( () =>    {console.log ("ok");} )
  .catch ( (err) => {console.error ("Fehler: ", err);} );
```

Src. 8: chmod in node: Promise API.

```
import fs from 'fs';
try {
  await fs.promises.chmod("file.txt", 0o775);
  console.log ("ok");
} catch (err) {console.error ("Fehler: ", err);}
```

Src. 9: chmod in node: Promise API mit await.

4. Events

Wie funktionieren Ereignisse?

Beispiel

```
var xhr = new XMLHttpRequest();
xhr.open("GET","http://bsp.com/daten.json");     // Objekt vorbereiten
xhr.addEventListener('load', function(e) {       // auf load Ereignis reagieren
  if (xhr.status >= 200 && xhr.status < 300) {  /* mit den Daten was tun */ }
  else { /* Fehlerbehandlung */}
});
xhr.send();  // Aufruf auslösen
```

Src. 10: Asynchrone Abfrage mit XMLHttpRequest

Quellen von Ereignissen

Typische Quellen:

- Benutzer-Interaktionen. Bsp: onmouseover
- Zustandswechsel bei asynchronen Aktivitäten. Bsp: Netzwerk-Abfrage
- Programmatisch erzeugte Ereignisse.
  Bsp: publish-subscribe, Ereignis-gesteuerte Programmierung.

Wichtige Schnittstellen:

- Events im Browser
- Events in Node
- Events in Browser Extensions

Typische Ziele:

- Ereignis wird an einen Target gesendet.

Entkoppelung durch Ereignisse

Räumliche Entkoppelung:

- Die interagierenden Komponenten müssen sich nicht kennen.

Semantische Entkoppelung:

- Schnittstellen müssen nicht aneinander angepaßt sein.
- Ereignis trägt alle Informationen, die Ereignisbehandlung ggf. brauchen könnte.
- Muß nicht auf Vorrat geschehen sondern nur Weg eröffnen, diese Informationen zu besorgen.

Synchronisations Entkoppelung

- Erzeugen und Bearbeiten von Ereignissen muß nicht in demselben Kontrollfluß (Thread, Prozeß) geschehen.

Beispiel für räumliche Entkoppelung: UI Entkoppelung

```
// Berechtigungen erwerben für ausführliche Enumeration
var devs = await navigator.mediaDevices.enumerateDevices();
var eve  = new CustomEvent ("discoveryCompleted", { detail: {devs} } );
document.body.dispatchEvent ( eve );  // programmatisch erzeugtes Ereignis
return devs;
```

Src. 11: Beispiel für UI Entkoppelung

- Library meines Audio Recorders für diese Vorlesung.
- Device discovery erfordert Vorbereitung hinsichtlich permissions.
- Discovery Komponente bietet zusätzlich `CustomEvent` an.
- Dadurch kann Info beliebig auf UI angezeigt werden.
- UI kann separat davon entwickelt werden.

Ereignisse behandeln

```
const btn = document.querySelector('button');
btn.onclick = function { alert ("clicked");}
```

Src. 12: Event-Handler als **Eigenschaften** von DOM-Elementen.

```
<button onclick="alert('clicked');">Klicken!</button>
<button onmouseover="myFunction(event);">Klicken!</button>
```

Src. 13: Event-Handler als **HTML-Attribute** von DOM-Elementen.

```
btn.addEventListener('click', myFunction);
...
btn.removeEventListener('click', myFunction);
```

Src. 14: Event Handler **hinzufügen** und **entfernen**.

Event Mechanismen

Default Verhalten:

- Manche Ereignisse haben ein default Verhalten.
- Bsp: Doppelklick im Text selektiert Wort.
- Das will man gelegentlich unterbinden: Dazu dient `preventDefault`.

Hierarchien:

- Manche Ereignisse entstehen in einer Hierarchie von Objekten (Bsp: DOM)
- **Erst Capture**:
  Ereignis wandert von Wurzel zum Zielobjekt des Ereignisses.
- **Dann Bubble**:
  Ereignis wandert vom Zielobjekt des Ereignisses zur Wurzel.
- Handler können Phasen-spezifisch registriert werden.
- Weiterwandern kann mit `stopPropagation` unterbunden werden.

Beispiel: Funktionsweise der Event Loop (1)

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "test.txt", true);        // initialisieren
xhr.onreadystatechange = function() {};   // event handler
xhr.send();                               // senden
```

Src. 15: Beispiel: Funktionsweise der Event Loop (1)

Analyse:

- Code ist völlig unproblematisch.

Funktionsweise der Event Loop (2)

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "test.txt", true);        // initialisieren
xhr.send();                               // senden
xhr.onreadystatechange = function() {};   // event handler
```

Src. 16: Beispiel: Funktionsweise der Event Loop (2)

Analyse:

- Der Event Handler wird immer aufgerufen
  obwohl er erst nach dem Aufruf gesetzt wird.
- Eine Race Condition tritt nie auf.
- Die Event Loop kann erst nach Beendigung des gesamten Threads
  wieder auf Ereignisse prüfen.

Funktionsweise der Event Loop (3)

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "test.txt", true);        // initialisieren
xhr.send();                               // senden
xhr.onreadystatechange = function() {};   // event handler
while (1) {}                              // nie fertig werden
```

Src. 17: Beispiel: Funktionsweise der Event Loop (3)

Analyse:

- Der Event Handler wird nie aufgerufen.
- Die Event Loop wird aufgrund der Schleife nie aufgerufen.

Funktionsweise der Event Loop (4)

```
var async = true;
var xhr = new XMLHttpRequest();
xhr.open('get', 'data.json', async);
xhr.send();

setTimeout(function delayed() { // race condition einbauen
  function listener() {         // wird das rechtzeitig genug aufgerufen???
    xhr.addEventListener('load', listener);
    xhr.addEventListener('error', listener);}
}, 3000);
```

Src. 18: Funktionsweise der Event Loop (4)

Event Loop Architekturen

Mechanismus:

- Runtime verwaltet eine Event Queue mit Ereignissen.
- Ereignisse werden eingetragen, wenn sie eintreffen.
- Wenn Thread *idle* wird, erfolgt nächste Abfrage der Event Loop.

Problem: Lost in continuation

```
var stub;
stub = getDataFromServerMachinery();
stub.onDataReady ( function (data) { // das ist der continuation handler
  var res1 = view.insert (data);
  var res2 = doNextStuff1 (res1);
  doNextStuff2 (res2);
});
// ähm...was soll hier geschehen?
// der **ganze** Rest gehört in den continuation handler
// bei mehreren asynchronen Aktivitäten wird das extrem verschachtelt
```

Src. 19: Der Kontrollfluß bei vielen verschachtelten asynchronen Aktivitäten wirkt sehr seltsam, da das **komplette weitere Programm** in die Ereignis-Behandlung eines Ereignisses gestopft werden muß.

5. Promises

Wie arbeiten Versprechungen?

Beispiel

```
var stub;
stub = getDataFromServerMachinery();
stub.then ( function (data) { return view.insert(data); } )
    .then ( doNextStuff1 )
    .then ( doNextStuff2 );
});
```

Src. 20: Arbeiten mit Promises

Teillösung:

- Auch hier steckt die Fortsetzung im `then` handler.
- Bisher mußte sie syntaktisch in callback / event handler stehen.
- Jetzt ist promise ein abstrakter stub, der weitergereicht werden kann.
- Damit kann das Hinzufügen weiterer Fortsetzungen woanders geschehen.

Fehlerbehandlung und Chaining

```
Request.get( 'http://google.com' )
  .then( function (e) { insert(e); return 17;  },  // fulfillment handler
         function (e) { showErr(e); return -2; })  // rejection handler
  .then( function (x) {console.info(x);} );
```

Src. 21: Fulfillment, Rejection und Chaining bei Promises

Vorgehen: Promise ruft auf:

- Im Erfolgsfall: Die erste Funktion sog. **fulfillment** of promise
- Im Fehlerfall: Die zweite Funktion sog. **rejection** of promise
- Welcher Fall Eintritt, weiß das Promise selber. Die Rückgabe-Werte der then-Methoden werden an die nächste Stufe der Pipeline weitergereicht

Ablauf

Konstruktion des Promise:

- dient nur dem Befüllen von Event Handlern
- wird sofort ausgeführt

Achtung:

- Promises bleiben bestehen, bis sie erfüllt oder zurückgewiesen wurden.
- Andere Promises im Hintergrund können dazwischen geraten.
- Race conditions zwischen mehreren Promises sind möglich.

Konstruktion von Promises

1. **Art:** Durch Promise-basierte APIs.
   - Alle neueren asynchronen API calls erzeugen ein Promise.
   - User muß sich nicht darum kümmern.

2. **Art:** Durch Promise Constructor
   - Erlaubt Wrapper um "alte", event oder callback basierte asynchrone API calls

Erzeugen eines Promise

```
new Promise (executor)
```

`executor` ist eine Funktion, die zwei selber wieder funktionale Argumente bekommt.

- 1. Argument: **resolve:** Wird auf fulfillment-Wert des Promise aufgerufen sobald die asynchrone Aktivität endet.
- 2. Argument: **reject:** Wird im Fehlerfall auf ein Fehlerobjekt aufgerufen.

Ausführung:

- `executor` wird sofort ausgeführt.
- Nachfolgende `then / catch` Kette wird ebenso sofort ausgeführt.
- Kette dient dem Befüllen geeigneter Handler-Slots im Promise.
- Letztlich also gleich wie callback-chain – nur nicht syntaktisch ersichtlich.

Beispiel: Transformation von Events auf Promises

```
function makeRequest (method, url) {  // generiert Promise
  return new Promise( function (resolve, reject) {
    var xhr = new XMLHttpRequest();
    xhr.open(method, url);
    xhr.onload = function () {
      if (this.status >= 200 && this.status < 300)
   {resolve(xhr.response); }
      else {reject({status: this.status}); }
    };
    xhr.onerror = function () { reject({status: this.status }); };
    xhr.send();   }); }

makeRequest('GET', 'http://example.com')              // Promise
  .then  ( function (val) { console.log(val);})      // befüllen
  .catch ( function (err) { console.error("Error!", err.statusText); } );
```

Src. 22: Transformation von Event-based auf Promise-based bei Browser XHR.

Beispiel: Transformation von Events auf Promises

```
import {readFile} from 'fs';

function readFilePromisified(filename) {
  return new Promise(
    function (res, rej) {
      readFile(filename, { encoding: 'utf8' },
        (err, data) => {
           if (err) {rej(err);} else {res(data);}
      });
  });  }

readFilePromisified("Filename")
  .then ( text   => {console.log(text);}  )
  .catch ( error => {console.log(error);} );
```

Src. 23: Transformation von Event-based auf Promise-based bei node readFile

Beispiel: Versprechen, etwas zu warten (= Alarm)

```
function delay(ms) {
  return new Promise( function (res, rej) {
    setTimeout(res, ms); });  }

delay(5000)
  .then(function () {console.log('5 Sekunden sind um')});
```

Src. 24: Versprechen, etwas zu warten (=Alarm)

Aufgaben: Promises

Geolocation:

- Bauen Sie die Geolocation API des Browsers aus dem obigen Beispiel so um,
- daß die API ein Promise zurückgibt.

Timeout:

- Bauen Sie die folgende Funktion:
- Gegeben sei ein beliebiges Promise-Objekt und eine Zahl.
- Zurückgegeben werde ein Promise-Objekt, das die gegebene Promise ausführt, aber mit Fehler abbricht, falls das länger als die gegebene Zahl Sekunden dauert.

Fork-Join Pattern

```
Promise.all([
    asyncFunc1(),  // erzeugt Promise als Rückgabewert
    asyncFunc2(),  // erzeugt Promise als Rückgabewert
  ])
.then(([result1, result2]) => {
 // erst wenn alle Werte da sind geht es weiter
  })
.catch(err => {
  // Durch erstes reject aktiviert
  });
```

Src. 25: Promise.all kann eine Art fork-join Mechanismus umsetzen

Beispiel: Fork Join Pattern

```
const fileUrls = [
    'http://example.com/file1.txt',
    'http://example.com/file2.txt',
];
const promisedTexts = fileUrls.map(httpGet);

Promise.all(promisedTexts)
.then(texts => {for (const text of texts) {console.log(text);}})
.catch(reason => {
    // Durch erstes reject aktiviert
});
```

Src. 26: Fork-Join Pattern

6. Asynchrone Funktionen

Noch eine kleine syntaktische Verschönerung für Promises.

Situation

Restproblem:

- Asynchrone Programmierung bleibt immer noch syntaktisch schwierig.
- Synchronisation erfordert spezifische Libraries oder gute Kenntnis der Event Loop.

Lösung:

- Erweitere Sprache um Konstruktion, die einem expliziten Warten entspricht.
- Funktion muß als `async` deklariert werden; warten durch `wait`.
- **Vorteil:** Code wird übersichtlicher.
- **Cave:** Kann in komplexerem Code zu Race Conditions führen
  da Verschachtelungen nicht mehr so deutlich sichtbar sind.
- **Cave:** Performanz kann wegen des Wartens sinken.

Beispiel mit Promise

```
fetch ('bild.jpg')
  .then(response => {
     if (!response.ok) {throw new Error("Fehler: " + response.status);}
     return response.blob(); })
  .then(myBlob => {
     let objectURL = URL.createObjectURL(myBlob);
     let image = document.createElement('img');
     image.src = objectURL;
     document.body.appendChild(image); })
  .catch(e => {console.log('Fehler in fetch: ' + e.message); });
```

Src. 27: Beispiel asynchroner Abfrage mit Promise.

Beispiel mit async / await

```
async function myFetch() {
  let response = await fetch ('bild.jpg');
  if (!response.ok) {throw new Error("Fehler: " + response.status);}
  let myBlob = await response.blob();
  let objectURL = URL.createObjectURL(myBlob);
  let image = document.createElement('img');
  image.src = objectURL;
  document.body.appendChild(image);
}

myFetch()
  .catch( e => {console.log(' Fehler in fetch: ' + e.message);} );
```

Src. 28: Beispiel asynchroner Abfrage mit async / await.

7. Parallele Verarbeitung

Hier: Mit parallelen Prozesse (statt mit Threads)

Kurzer Überblick

Single-Threaded Programmierung:

- In Node und Browser mit klassischer Event Loop.

Multi-Threaded Programmierung

- Worker-Schnittstelle.
- Kein gemeinsamer Speicher.
- Programmier-Modell des Message Passing.
- Runtime kann auf Prozesse oder (Hyper)-Threading abbilden.
- Keine Multi-Processing Schnittstelle..

Multi-Prozeß Programmierung

- Meint hier jetzt die klassische OS Multi-Prozeß Schnittstelle.
- Nur in Node verfügbar.

Beispiel: Parent

```
// parent.js
var child_process = require('child_process');
var numchild      = require('os').cpus().length;
var done          = 0;

for (var i = 0; i < numchild; i++){
  var child = child_process.fork('./child.js');
  child.send((i + 1) * 1000);
  child.on('message', function(message) {
    console.log('[parent] received message from child:', message);
    done++;
    if (done === numchild) {
      console.log('[parent] received all results');
    }
  });
}
```

Beispiel: Child

```
// child.js
process.on('message', function(message) {
  console.log('[child] received message from server:', message);
  setTimeout(function() {
    process.send({
      child  : process.pid,
      result  : message + 1
    });
    process.disconnect();
  }, (0.5 + Math.random()) * 5000);
});
```

Src. 30: Parallele Prozesse in Node: Child

8. Animationframe

Wichtige Technik für Animationen.

Eng verwandt mit `setTimeout` und `setInterval`.

Problem

Was tun mit Aufgaben, die

- `document` benötigen (also kein Web Worker)
- nicht den UI Thread blockieren dürfen
- regelmäßig laufen müssen.

Typischer Use Case: Animation

Erste Idee: setTimeout / setInterval

- Problem: Feste Zeittaktung kann falsch sein.
- Aufgabe läuft zu selten – Animation ruckelt.
- Aufgabe läuft zu oft – Andere Aufgaben werden ausgebremst.
- Problem: Dynamische Anpassung kann nur Vergangenheit berücksichtigen.

Beispiel

```
var start   = null;
var element = document.getElementById('ElementToAnimate');
element.style.position = 'absolute';
function step (timestamp) {
  if (!start) {start = timestamp;}
  var prog = timestamp - start;
  element.style.left = Math.min(prog / 10, 200) + 'px';
  if (prog < 2000) {window.requestAnimationFrame(step);}
}
window.requestAnimationFrame(step);
```

Src. 31: Request Animation Frame

Lösung

`requestAnimationframe`:

- Fordert Aufruf einer Funktion an, wenn das System gerade Zeit hat.
- Übergibt präzisen Zeitstempel, damit Funktion Aufrufzeitpunkt kennt.
- Zeitstempel ermöglicht Berechnung des erforderlichen Animations-Fortschritts,
- selbst wenn Aufruf mal später oder früher kommt.
- System kann lastabhängig & dynamisch selber steuern, wann Thread übergeben wird.

Abgrenzung zu `setTimeout (fct, 0)`:

- `setTimeout` erlaubt auch wiederholte Task-Unterbrechung.
- erlaubt aber zu häufigen Task-Aufruf.
- `requestAnimationFrame` koordiniert noch mit dem Frame-Rate des Systems.

Literaturhinweis: Übersicht

9. Web & Service Worker

Behandeln wir in einem separaten Abschnitt.

Anhang

Übersicht

Programmquellenverzeichnis (1/3)

Rechtliche Hinweise (1)

Die hier angebotenen Inhalte unterliegen deutschem Urheberrecht. Inhalte Dritter werden unter Nennung der Rechtsgrundlage ihrer Nutzung und der geltenden Lizenzbestimmungen hier angeführt. Auf das Literaturverzeichnis wird verwiesen. Das Zitatrecht in dem für wissenschaftliche Werke üblichen Ausmaß wird beansprucht. Wenn Sie eine Urheberrechtsverletzung erkennen, so bitten wir um Hinweis an den auf der Titelseite genannten Autor und werden entsprechende Inhalte sofort entfernen oder fehlende Rechtsnennungen nachholen. Bei Produkt- und Firmennamen können Markenrechte Dritter bestehen. Verweise und Verlinkungen wurden zum Zeitpunkt des Setzens der Verweise überprüft; sie dienen der Information des Lesers. Der Autor macht sich die Inhalte, auch in der Form, wie sie zum Zeitpunkt des Setzens des Verweises vorlagen, nicht zu eigen und kann diese nicht laufend auf Veränderungen überprüfen.

Alle sonstigen, hier nicht angeführten Inhalte unterliegen dem Copyright des Autors, Prof. Dr. Clemens Cap, ©2020. Wenn Sie diese Inhalte nützlich finden, können Sie darauf verlinken oder sie zitieren. Jede weitere Verbreitung, Speicherung, Vervielfältigung oder sonstige Verwertung außerhalb der Grenzen des Urheberrechts bedarf der schriftlichen Zustimmung des Rechteinhabers. Dieses dient der Sicherung der Aktualität der Inhalte und soll dem Autor auch die Einhaltung urheberrechtlicher Einschränkungen wie beispielsweise Par 60a UrhG ermöglichen.

Die Bereitstellung der Inhalte erfolgt hier zur persönlichen Information des Lesers. Eine Haftung für mittelbare oder unmittelbare Schäden wird im maximal rechtlich zulässigen Ausmaß ausgeschlossen, mit Ausnahme von Vorsatz und grober Fahrlässigkeit. Eine Garantie für den Fortbestand dieses Informationsangebots wird nicht gegeben.

Die Anfertigung einer persönlichen Sicherungskopie für die private, nicht gewerbliche und nicht öffentliche Nutzung ist zulässig, sofern sie nicht von einer offensichtlich rechtswidrig hergestellten oder zugänglich gemachten Vorlage stammt.

Use of Logos and Trademark Symbols: The logos and trademark symbols used here are the property of their respective owners. The YouTube logo is used according to brand request 2-9753000030769 granted on November 30, 2020. The GitHub logo is property of GitHub Inc. and is used in accordance to the GitHub logo usage conditions https://github.com/logos to link to a GitHub account. The Tweedback logo is property of Tweedback GmbH and here is used in accordance to a cooperation contract.

Rechtliche Hinweise (2)

Disclaimer: Die sich immer wieder ändernde Rechtslage für digitale Urheberrechte erzeugt für mich ein nicht unerhebliches Risiko bei der Einbindung von Materialien, deren Status ich nicht oder nur mit unverhältnismäßig hohem Aufwand abklären kann. Ebenso kann ich den Rechteinhabern nicht auf sinnvolle oder einfache Weise ein Honorar zukommen lassen, obwohl ich – und in letzter Konsequenz Sie als Leser – ihre Leistungen nutzen.

Daher binde ich gelegentlich Inhalte nur als Link und nicht durch Framing ein. Lt EuGH Urteil 13.02.2014, C-466/12 ist das unbedenklich, da die benutzten Links ohne Umgehung technischer Sperren auf im Internet frei verfügbare Inhalte verweisen.

Wenn Sie diese Rechtslage stört, dann setzen Sie sich für eine Modernisierung des völlig veralteten Vergütungssystems für urheberrechtliche Leistungen ein. Bis dahin klicken Sie bitte auf die angegebenen Links und denken Sie darüber nach, warum wir keine für das digitale Zeitalter sinnvoll angepaßte Vergütungssysteme digital erbrachter Leistungen haben.

Zu Risiken und Nebenwirkungen fragen Sie Ihren Rechtsanwalt oder Gesetzgeber.

Weitere Hinweise finden Sie im Netz hier und hier oder hier.

Zitierweise dieses Dokuments

Wenn Sie Inhalte aus diesem Werk nutzen oder darauf verweisen wollen,
zitieren Sie es bitte wie folgt:

Clemens H. Cap: Asynchrone Programmierung. Electronic document. https://iuk.one/1066-1007
1. 6. 2021.

Bibtex Information: https://iuk.one/1066-1007.bib

```
@misc{doc:1066-1007,
   author        = {Clemens H. Cap},
   title         = {Asynchrone Programmierung},
   year          = {2021},
   month         = {6},
   howpublished  = {Electronic document},
   url           = {https://iuk.one/1066-1007}
}
```

Typographic Information:
Typeset on June 1, 2021
This is pdfTeX, Version 3.14159265-2.6-1.40.21 (TeX Live 2020) kpathsea version 6.3.2
This is pgf in version 3.1.5b
This is preamble-slides.tex myFormat©C.H.Cap

Verzeichnis aller Folien

Legende:
- Fortsetzungsseite
- Seite ohne Überschrift
- Bildseite