lacktime

issue 1

Eager vs. Lazy

Agenda

- Abstraktion der grundsätzlichen Problematik
  - Terminologie
  - Erfassung der Probleme bzw. Entscheidungspunkte
    - Performance vs Einfachheit
    - Replikationsproblem
- Umsetzung der Abstraktion auf Web-Applikationen
  - Umsetzung auf lokale bzw. entfernte Funktionen
- Ausblick: Können Promises zu anderen Trade-Offs führen?
- Ausblick: Zirkuläre Datenstrukturen

Abstraktion

Die Begriffe Eager und Lazy kommen von der Applikation von Funktionen (die Evaluierung von Funktionsaufrufen).

Gedanken an aktuelle Probleme und Lösungsvorschläge blockieren den freien Gedankengang.

=> Zunächst betrachten wir keine Web-Applikation sondern lediglich die Applikation von Funktionen.

Terminologie - Eager (Call by Value)

```
function f (a, b, c) {
   return a + b + c;
}
eval ( f(3, 4 * 2, sqrt(9)) )
= eval ( f(3, 8, sqrt(9)) )
= eval ( f(3, 8, 3) )
= eval ( 3  + 8 + 3 )
= eval ( 14 ) = 14
```

Terminologie - Lazy (Call by Name)

Lazy:

```
function f (a, b, c) {
   return a + b + c;
}
eval ( f(3, 4 * 2, sqrt(9)) )
= eval ( 3 + 4 * 2 + sqrt(9) )
= eval ( 3 + 8 + sqrt(9) )
= eval ( 3  + 8 + 3 )
= eval ( 14 ) = 14
```

Terminologie

Eager vs. Lazy

- Gibt es einen Unterschied?
- Performance-Unterschiede?
- Beispiel?

Terminologie - Eager (Call by Value)

```
function f (a, b) {
   return a || b + b;
}
eval ( f(3, fib(40)) )
= eval ( f(3, 102334155) )
= eval ( 3  || 102334155 + 102334155 )
= eval ( 3 ) = 3
```

Terminologie - Lazy (Call by Name)

```
function f (a, b) {
   return a || b + b;
}
eval ( f(3, fib(40)) )
= eval ( 3 || fib(40) + fib(40)) )
= eval ( 3 ) = 3
```

Terminologie - Teueres Statement

Ein Statement, das

- nicht benötigte Ergebisse berechnet oder
- wesentlich mehr Daten als nötigt beschafft

wird teueres Statement genannt.

Terminologie - Eager (Call by Value)

```
function f (a, b) {
   return a || b + b;
}
eval ( f(0, fib(40)) )
= eval ( f(0, 102334155) )
= eval ( 0  || 102334155 + 102334155 )
= eval ( 102334155 + 102334155 )
= eval ( 204668310 ) = 204668310
```

Terminologie - Lazy (Call by Name)

```
function f (a, b) {
   return a || b + b;
}
eval ( f(0, fib(40)) )
= eval ( 0 || fib(40) + fib(40)) )
= eval ( fib(40) + fib(40)) )
= eval ( 102334155 + fib(40)) )
= eval ( 102334155 + 102334155) )
= eval ( 204668310 ) = 204668310
```

Terminologie - Lazy (Call by Need)

```
function f (a, b) {
   return a || b + b;
}
```

eval (f(0, fib(40)))
= eval (0 || fib(40) + fib(40)))
= eval (fib(40) + fib(40)))
= eval (102334155 + fib(40))) => merke eval(fib(40)) = 102334155
= eval (102334155 + 102334155)) => schnelle Evaluierung
= eval (204668310) = 204668310

Entscheidung: Eager oder Lazy?

```
function f (a, b, c, d, e, g, h, i, j, k, l, m, n, o, p, q, r, s, t) {
  if (a < 2) return b + b;
  if (a < 4) return c + c;
  if (a < 6) return d + d;
  if (a < 8) return e + g;
  if (a < 10) return g + h;
  if (a < 12) return i + j + k;
  if (a < 14) return l + m + n;
  if (a < 16) return o + p + q;
  return r + s + t;
}
```

Entscheidung: Eager oder Lazy?

```
function f (a, b, c, d, e, g, h, i, j, k, l, m, n, o, p, q, r, s, t) {
   if (a < 2) return b + b;
   if (a < 4) return c + c;
   if (a < 6) return d + d;
   if (a < 8) return e + g;
   if (a < 10) return g + h;
   if (a < 12) return i + j + k;
   if (a < 14) return l + m + n;
   if (a < 16) return o + p + q;
   return r + s + t;
}
```

Im Beispiel mit fib(40) gilt:

- Call-By-Name: ca. 7,4 mal schneller
- Call-By-Need: ca. 8,6 mal schneller

Statistiken

- 4844 Datentypen, 45519 Struktur-Felder
- Konnektoren pro App (avg: 3.5, max: 27)
- Es gibt 18935 Calls auf DCMR
- 537 unterschiedliche Datentypen
- Calls pro Konnektor (avg: 9.5, max: 347)
- Parameter pro Call (avg: 5.4, max: 44)
- 35.2 Call-Parameter haben den gleichen Datentyp
- 3.5 * 9.5 * 5.4 / 35.2 = 5.1 Datentypen pro App bzgl. Konnektoren

Statistiken

- 4844 Datentypen, 45519 Struktur-Felder
- 683 verschiedene Datentypen über alle Call-, SBO-, CBO-Parameter
- SBOs pro App (avg: 10.3, max: 115)
- Functions pro SBO (avg: 7.5, max: 136)
- 43,3 Function-Parameter haben den gleichen Datentyp

Statistiken

- CBOs pro App (avg: 10.1, max: 146)
- Functions pro CBO (avg: 3.2, max: 66)
- Konnektoren pro App (avg: 3.5, max: 27)
- Es gibt 18935 Calls auf DCMR
- 537 unterschiedliche Datentypen
- Calls pro Konnektor (avg: 9.5, max: 347)
- Parameter pro Call (avg: 5.4, max: 44)
- 35.2 Call-Parameter haben den gleichen Datentyp
- 3.5 * 9.5 * 5.4 / 35.2 = 5.1 Datentypen pro App bzgl. Konnektoren

Performance - Wartezeiten

Eager (Call By Value):

- Einmal relativ große Wartezeit. Je nach Anwendung im Sekundenbereich.

Lazy:

- Viele kleine Wartezeiten im 100-ms-Bereich.

Relation:

- Einmal blinzeln dauert zwischen 100 und 400 ms.

Aufwand

Eager (Call By Value):

- Werte sind einfach zu handhaben.

Lazy:

- Das Reihenfolge-Problem führt grundsätzlich zu höherem Aufwand.
- Sneak-Peak in Richtung Web-App:
  - asynchrone Aufrufe
  - Verzögerte Ausführung mit Hilfe von Funktionen (Callback-Hell)
  - Blockierungsaufwand in UI (Reihenfolge-Problem)

Aktualität - Replikationsproblem

```
function f (lazyY) {
    return lazyY - lazyY;
}
```

Gilt im Allgemeinen lazyY - lazyY = 0 ???

Aktualität - Replikationsproblem

Wenn Zustände (mutable Objects) lazy angefragt werden, werden wir aktuellere Zustände erhalten.

Wenn wir den Zustand eines Objektes mehrmals anfragen führt dies zu unterschiedlichen Ergebnissen.

Fall 1: Ich möchte immer den aktuellsten Zustand haben.

Fall 2: Zur Konsistenzbedingung meiner Funktion gehört, dass ich mit dem selben Zustand arbeite.

Aktualität - Replikationsproblem

Eager:

- Ich hole mir frühzeitig alle Zustände => hohes Alter aller Zustände.
- Alle Zustände sind gleich alt.
- Innerhalb meiner Funktion gilt immer Zustand1 = Zustand1

Call By Name:

- Meine Zustände sind immer aktuell.
- Innerhalb meiner Funktion kann Zustand1 != Zustand1 gelten.

Aktualität - Replikationsproblem

Call By Need:

- Meine Zustände sind beim ersten benötigten Zugriff aktuell.
- Das Alter der benötigten Zustände ist unterschiedlich, aber aktueller im Vergleich zu Eager.
- Innerhalb meiner Funktion gilt immer Zustand1 = Zustand1

Analogie zu Web-Applikationen

Eager:

- Model-Initialisierung: Es gibt eine Initialisierungsphase in der alle benötigten Daten vom Server bereitgestellt werden.
- Während der User-Interaktion wird nur mit den lokalen Daten im Model interagiert.
- => Das Model enthält in der Regel ein Vielfaches der Daten die benötigt werden.
- => Die Initialisierung benötigt viel Zeit.
- => Die Interaktion nach der Initialisierungsphase ist schnell
- => Daten sind nicht die aktuellsten. Refresh-Problem.

Analogie zu Web-Applikationen

Lazy:

- Model-Initialisierung: Leeres Model oder nur Overview-Daten.
- Während der User-Interaktion wird das Model befüllt.
- => Das Model enthält nur Daten, die der User benötigt hat.
- => Die schrittweise Modelbefüllung benötigt wenig Zeit.
- => Die Interaktion benötigt asynchrone Funktionsaufrufe.
- => Daten können beliebig aktuell sein.
  - Zugriff auf Model entspricht Call By Need.
  - Ständiger Backend-Request entspricht Call By Name.

Analogie zu Web-Applikationen

Beispiel Master-Detail für Kundenaufträge:

- Eine Liste zeigt alle Kunden.
  - Die Liste benötigt weder alle Aufträge jedes Kunden, noch die Kenntnis, wieviele Haustiere der Kunde hat.
  - Es werden nur soviel Daten benötigt, dass der User seine Auswahl treffen kann.
- Eine Detail-Anzeige wird erst nach Auswahl angezeigt.
  - Eine Detail-Anzeige zeigt die Liste aller Aufträge des Kunden.
  - Die Detail-Anzeige zeigt die Lieferadresse des Kunden an.

Analogie zu Web-Applikationen

Beispiel Master-Detail für Kundenaufträge:

- Es gibt 1000 Kunden
- Durchschnittlich gibt es 10 Aufträge pro Kunden.
- Zu jedem Kunden gibt es eine Rechnungs- und eine Lieferanschrift.
- Der User benötigt Informationen zu 10 Kunden.
- Kundenliste benötigt nur Kundennummer und -name.

Analogie zu Web-Applikationen

Beispiel Master-Detail für Kundenaufträge:

- Eager:
  - Für die Kundenliste werden alle Adressen und Auftäge geladen.
  - Keine Zeitverzögerung zwischen Auswahl und Detail-Anzeige.
  - 12000 Dateneinheiten (1000 * (10 + 2)) werden in der Initialisierungsphase geholt.
  - 1060 Dateneinheiten (1000 + 60) werden benötigt.
  - Der letzte Auftrag des 5. Kunden wird nicht angezeigt, da er erst vor 5 Minuten abgeschlossen wurde.

Analogie zu Web-Applikationen

Beispiel Master-Detail für Kundenaufträge:

- Lazy:
  - Für die Kundenliste werden nur Kundennummern und Kundennamen geladen 1000.
  - Die 5 Selektionen führen zu 5 Anfragen à 12 (10+2) Dateneinheiten.
  - 1060 Dateneinheiten (1000 + 60) werden benötigt.
  - Der letzte Auftrag des 5. Kunden wird angezeigt, da die Auswahl vor 3 Minuten stattfand.
  - Zeitverzögerung zwischen Auswahl und Detail-Anzeige

Callback-Hell

```
function showDetail (auswahl) {
  var cbShowDetailView = function (auswahl, orders) {
    releaseView();
    showDetailView(auswahl, adressen, orders);
  };
  var cbGetOrders = function (auswahl, adressen) {
    getOrders(auswahl, cbShowDetailView);
  };
  blockView();
  getAddresses(auswahl,cbGetOrders);
}
```

Callback-Hell

```
function showDetail (auswahl) {
  var cbShowDetailView = function (auswahl, orders) {
    releaseView();
    showDetailView(auswahl, adressen, orders);
  };
  var cbGetOrders = function (auswahl, adressen) {
    getOrders(auswahl, cbShowDetailView);
  };
  blockView();
  getAddresses(auswahl,cbGetOrders);
}
```

Callback-Hell

Code-Komplexität vs. Performance

- Wartbarkeit ist eine wichtige Design-Entscheidung.
- Je nach Service-Level-Agreement und Programmier-Know-How wird die Entscheidung zu Gunsten von Eager ausfallen.

Promises

```
function showDetail (auswahl) {
   getAddresses(auswahl)
      .then(getOrders)
      .then(showDetailView);
}
```

Zirkuläre Datentypen

- Eager:
  => führt unweigerlich zu Endlosschleifen
- Lazy:
  => kann Datentypen in beliebiger tiefe auflösen