www.javaspektrum.de

Ausgabe 4/2021
D € 9,90 · A € 10,90 · CHF 18,20

4 194156 109901 04

JavaSPEKTRUM

Magazin für professionelle Entwicklung und digitale Transformation

Think big –
Datenmengen intelligent verarbeiten

Interview

Die Doktoren Rainer Seßner und Susan Lindner von Bayern Innovativ über das Vorantreiben neuer Technologien und ihre Wirkung in der Digitalisierung

Data Lakes in der Cloud – ein Lösungsansatz

Big Data im Kleinen: Einsatz von ML auf IoT-Geräten

Fachthemen

Grundlagen und Best Practices aus dem Cloud Performance Engineering

AsyncAPI: Verteilte asynchrone Kommunikation für mehr Skalierbarkeit

zum Inhalt

Think big – Datenmengen intelligent verarbeiten

G30325

Für große & kleine Mengen

Der Einsatz von Zufallsbezeichnern in der Softwareentwicklung und im Alltag

Christian Heitzmann

Wieso müssen ID-Nummern eigentlich immer sequenziell sein, URL-Pfade stets der gleichen Systematik folgen und Exception-Messages sinnfreien Text enthalten, der im Ernstfall sowieso nicht weiterhilft? Ist es nicht eher so, dass die Antwort „Weil man es halt so macht" irreversible Abhängigkeiten einführt und langfristigen Mehraufwand verursacht? Spielen wir doch einmal den ungewöhnlichen Ansatz durch, in der Softwareentwicklung wie auch im Alltag konsequent auf Zufallsbezeichner zu setzen.

Wer als regelmäßiger JavaSPEKTRUM-Leser meine Artikel kennt, wird wahrscheinlich schon bemerkt haben, dass die Links auf die herunterladbaren Quellcodes jeweils leicht „kryptisch" anmuten. Betrachten wir zum Beispiel meinen letzten Artikel aus JavaSPEKTRUM 2/2021 mit dem Titel *„Future und CompletableFuture"* [HE4L]. Dort wurde für den Quellcode die URL *https://link.simplexacode.ch/va9s* angegeben. Ebenfalls findet sich im Artikel der Link *https://link.simplexacode.ch/ewvs* auf einen meiner Blogeinträge zum Thema Floating-Point Numbers.

Zum Zeitpunkt der Drucklegung verweist der erste Link auf *https://www.simplexacode.ch/source-code/javaspektrum_2021_2___completablefuture_2021.02.zip*. Hätte ich diesen Link allerdings so abdrucken lassen, wäre es a) für die Leser nicht nur viel umständlicher gewesen, ihn abzutippen, sondern b) bliebe der Link auf Ewigkeiten so eingefroren und hätte auch für immer entsprechend bewirtschaftet werden müssen. Mein eigener URL-Shortener hingegen ist denkbar einfach und auf dem Apache-Webserver meiner Firma nichts anderes als eine manuell gepflegte Tabelle mit HTTP-Redirects in der Datei `.htaccess`. Gleichzeitig ist sein Nutzen enorm: So steht es meiner SimplexaCode frei, eines Tages beispielsweise das Verzeichnis `source-code/` in `sourcecode/` oder `quellcode/` umzubenennen, eine komplett neue Dateihierarchie umzusetzen oder auch nur eine neue korrigierte Version der Quellcodes hochzuladen (dann zum Beispiel mit dem Suffix `_2021.08.zip`, weil prinzipiell keine Dateien ohne Versionskennung herausgehen).

Die entscheidende Überlegung ist, dass es auch ohne URL-Shortener einer solchen Redirect-Tabelle bedarf, nämlich sobald die erste Änderung ansteht. Wurde in der Vergangenheit mit Nachteil das fixe Verzeichnis `source-code/` abgedruckt oder anderweitig veröffentlicht, soll dieses aber neu `quellcodes/java/` lauten, dann müssten Redirects von `source-code/` auf `quellcodes/java/` programmiert werden. Solche Tabellen werden spätestens bei der zweiten Änderung sehr schnell sehr hässlich, und das Risiko, einen früher mal extern veröffentlichten Link zu vergessen, ist relativ hoch. Werden hingegen ausschließlich Short-URLs herausgegeben, ist ein solches Refactoring denkbar einfach: Das „Grooming" der Redirect-Tabelle ist trivial, und der Aufrufer ist nicht ansatzweise betroffen.

Unabdingbarkeit des Zufalls

Ein wichtiges Kriterium ist, dass die Short-URLs konsequent aus einem richtigen (Pseudo-)Zufallsgenerator stammen. Wie diese erzeugt werden, sehen wir gleich weiter unten. Alternativen zu Zufallsgeneratoren sind hingegen nicht überzeugend:

- **Zufallsbezeichner aus dem Kopf:** Der Mensch ist ein denkbar schlechter Zufallsgenerator. Er würde zum Beispiel das Kürzel `ente` als viel weniger „zufällig" betrachten als `qypz`, da Letzteres „doch viel mehr komische Buchstaben" hat. Dabei sind beide Kürzel absolut gleichwahrscheinlich.
- **Zufallszeichen mit Hand auf die Tastatur klatschen:** Probieren Sie einmal, auf diese Art 100 Dateien mit jeweils vier zufälligen Buchstaben zu benennen. Ich wette, spätestens ab der 20. Datei kommt es im gemeinsamen Ordner zum ersten Namenskonflikt – und wahrscheinlich heißt dieser sogar `asdf`.
- **Sprechende Abkürzungen:** Hätte demnach mein letzter JavaSPEKTRUM-Quellcode zum Thema *„Future und CompletableFuture"* das Kürzel `fucf` haben sollen? Oder doch eher `cofu`? Oder `futu`? Oder `cplf`? Es ist nur eine Frage der Zeit, bis es auch hier zum ersten Konflikt käme. Das kann wahrscheinlich jeder Leser, der schon einmal in einer Firma mit mehr als 30 Angestellten tä-

Christian Heitzmann ist Java- und Python-zertifizierter Softwareentwickler mit einem CAS in Machine Learning und Inhaber der Simplexa-Code AG in Luzern. Er entwickelt seit über 20 Jahren Software und unterrichtet bzw. doziert seit 10 Jahren unter anderem im Bereich der Java- und Python-Programmierung, Mathematik und Algorithmik.
E-Mail: christian.heitzmann@simplexacode.ch

tig war, bei den historischen (zweistelligen) Mitarbeiterkürzeln bezeugen. Sofern eine Systematik überhaupt jemals vorhanden war, geht sie ab der ersten Kollision verloren, und die Kürzel müssen schlussendlich doch wieder in einer Tabelle nachgeschaut und gepflegt werden.

- **Sequenzielle Bezeichner:** Hätte in diesem Fall mein erster Shortlink nach Firmengründung `aaaa` (oder `0001`), und der Shortlink auf die Quellcodes des letzten Artikels `aaaq` resp. `0017` heißen müssen? Welche ungewollte Information wird mitgeliefert, wenn der Leser so erkennen kann, dass dies erst (oder schon?) der 17. Shortlink ist? Wie schwierig ist es, die anderen 16 anzuschauen? Macht der Autor das bei seinen Rechnungen auch so, dass er im Sommer 2021 eine Rechnung mit der Nummer 3 verschickt (oder andersherum eine Rechnung mit der Nummer 6521)?

UUIDs

Bevor wir weitere Anwendungsbeispiele für den Einsatz von Zufallsbezeichnern ansehen, müssen wir zuerst klären, wo wir solche zufälligen IDs einfach herbekommen. Die meisten erfahrenen Softwareentwickler werden wohl schon mit *Universally Unique Identifiers (UUIDs)* Bekanntschaft gemacht haben. Das sind netto 32-stellige Hexadezimalzahlen, aufgeteilt in fünf Gruppen, die beispielsweise so aussehen: `3ecc17f8-d9c7-4de6-8b47-efb1447afd94`

Variante	Generatorzeichensatz	Beispiele

nur Großbuchstaben	ohne `0DO`, `1I`, `2Z`, `5S`, `8B`	`4NQ9`
		`XUYL-ALEU`
		`AN6P-R3YR-N7HC-XAW3`

nur Kleinbuchstaben	ohne `1l`	`h8d4`
		`3e6b-nms2`
		`s7ng-axy2-ocsb-szqz`

Groß- **und** Kleinbuchstaben (d. h. der Zufallsbezeichner besteht sowohl aus Groß- als auch Kleinbuchstaben)	ohne `0DO`, `1Il`, `2Z`, `5S`, `8B`, `Kk`	`pXm4`
		`6x49-wfMX`
		`AMh9-vX3Q-WCwP-r7Ya`

Groß- **oder** Kleinbuchstaben (d. h. der Zufallsbezeichner wird komplett in Groß- oder Kleinbuchstaben dargestellt, was aber vorab nicht einschränkend festgelegt wird)	ohne `0DO`, `1IL`, `2Z`, `5S`, `8B` (in Großbuchstabendarstellung)
	ohne `0do`, `1il`, `2z`, `5s`, `8b` (in Kleinbuchstabendarstellung)
		`XTYW`
		`KJEH-RKV6`
		`7ARN-336F-YHHF-CVY9`
		`xtyw`
		`kjeh-rkv6`
		`7arn-336f-yhhf-cvy9`

Tabelle 1: Varianten und Beispiele je nach Groß- und Kleinschreibung

In Java lassen sich derartige zufällige UUIDs ganz einfach mit dem Befehl `UUID.randomUUID()` erzeugen [3LPH]. Während Version 1 der UUID-Spezifikation noch MAC-Adresse und Zeitstempel des Erzeugers enthielt – was meines Erachtens inakzeptabel ist – basiert die heute vorherrschende Version 4 rein auf Zufallswerten [EXFQ]. Nichtsdestotrotz ist ihr Einsatz in der Interaktion mit Menschen sperrig, da die IDs nicht nur viel zu lang, sondern auch schwierig zu lesen, zu kommunizieren und zu vergleichen sind.

Benutzerfreundliche und flexible Zufallsbezeichner

Ich habe mir daher bereits vor über 3 Jahren zum Zeitpunkt meiner Firmengründung überlegt, wie Zufallsbezeichner benutzerfreundlicher und flexibler generiert werden können. Folgende Überlegungen lagen dem Design dieser Zeichenketten zugrunde:

- Sie bestehen nur aus Ziffern und lateinischen Buchstaben.
- Sie werden stets als Viererblock erzeugt. Längere Zufallszeichenketten entstehen durch Aneinanderreihung solcher Viererblöcke, die dann durch Bindestrich-Minus (dem einzigen Sonderzeichen) voneinander getrennt werden.
- Von Menschenaugen einfach zu verwechselnde Ziffern und Buchstaben werden aus dem Generatorzeichensatz entfernt.
- Dabei gibt es je nach geplantem Einsatzzweck vier Varianten bzgl. der Groß- und Kleinschreibung (s. Tabelle 1).

Die Gruppierung in Viererblöcke basiert auch auf den Erkenntnissen der kognitiven Psychologie. Die berühmte *Millersche Zahl* besagt, dass das Kurzzeitgedächtnis 7 ± 2 Informationseinheiten (Chunks) verarbeiten bzw. präsent halten kann [AAPH]. Dieser Wert – immerhin aus dem Jahre 1956 – gilt aber heute als veraltet. Die aktuelle Forschung spricht von 4 bis maximal 5 Chunks [PNQF]. Das kann man auch ganz einfach an sich selber beobachten: Ein Viererblock der Kreditkarte lässt sich relativ einfach fürs Abtippen merken, Fünferblöcke, wie sie zum Beispiel bei Referenznummern auf Schweizer Einzahlungsscheinen vorkommen, sind hingegen schon merklich unangenehmer.

Als Service an Sie, liebe Leserin und Leser, finden Sie unter [FVEK] den Java-Quellcode zur Generierung beschriebener Zufallsbezeichner. Zusätzlich steht Ihnen unter [PKC4] eine Website zur Verfügung, auf der Sie mit nur einem Klick jeweils einen neuen Zufallsdatensatz analog den Beispielen in Tabelle 1 erhalten. Auf jener Seite steht auch beschrieben, wie sich der Webservice zur maschinellen Abfrage benutzen lässt.

Anwendungsbeispiele in der Softwareentwicklung

Die folgenden drei Beispiele zeigen auf, wie Zufallsbezeichner in der Softwareentwicklung eingesetzt sein können.

Logeinträge, Exception- und Error-Messages

Hand aufs Herz: Wer beschränkt sich bei der Analyse großer Logdateien letzten Endes nicht auf den Klassen- und Methodennamen und die Zeilennummer des Fehlers, um ab diesem Punkt die Suche direkt im Quellcode fortzusetzen? Rechtfertigt sich der Aufwand wirklich, vermeintlich „sprechende“ Logeinträge, Exception- und Error-Messages zu verfassen? Erfahrungsgemäß wird deren Generierung von Entwicklern als lästige und unwichtige Arbeit empfunden, was sich dann auch in der orthografischen und grammatikalischen Qualität ebensolcher Nachrichten widerspiegelt; von einem einheitlichen Format ganz zu schweigen.

Listing 1 zeigt zwei Beispiele einer auf Zufallsbezeichnern basierenden Alternative. Tritt ein Fehler auf, ist man mit einer Volltextsuche nach `"Exception Code WR7U"` bzw. `"Error Code CREA"` sofort an der richtigen Stelle im Code. Da das Erstellen derartiger Nachrichten deutlich einfacher ist, motiviert das gleichzeitig auch zu vermehrten Logeinträgen und Zustandsprüfungen gemäß dem Fail-fast-Prinzip; und das wiederum kommt der Codequalität viel mehr zugute, als seltene, aber ausführliche Log- und Fehlermeldungen.

```
Objects.requireNonNull(value, "Exception Code WR7U");

if (!Modifier.isPublic(modifiers)) {
  throw new AssertionError("Error Code CREA");
}
```

Listing 1: Java-Exception und -Error mit Zufallsbezeichnern

HTML-IDs und JavaScript

Wer viel in HTML und JavaScript programmiert, kennt die „ID-Flut". Jedes HTML-Element muss mit einer eindeutigen ID versehen werden, um es skriptseitig ansprechen zu können. Sie ahnen es schon: Systematische selbstsprechende Bezeichner brauchen sehr viel kognitive Energie, und die Frage ist, ob sich dieser Aufwand wirklich lohnt. Denn systematische Bezeichner sind nur dann sinnvoll, wenn sie ohne Anstrengung und eindeutig (re-)konstruiert werden können. Wenn eine Element-ID letzten Endes doch jeweils wieder nachgeschaut werden muss, um herauszufinden, worauf sie sich bezieht, dann kann sie auch gleich direkt mit einem Zufallsbezeichner versehen werden.

Listing 2 zeigt ein derartiges Beispiel. Entscheidend ist auch hier, dass eine Volltextsuche im Quelltext nach dem Zufallsbezeichner `"ia2n"` sicher schneller ans Ziel führt als zum Beispiel `"examTopicsProblemlLearningObjectives"` oder gar `"exTopsProblLearnObjs"`. Und wenn Aufgabe 1 später zu Aufgabe 5 werden soll, dann bleibt der Zufallsbezeichner hiervon völlig unberührt.

```
<div class="learning_objectives" id="ia2n">
  <h2>Lernziele | Aufgabe 1</h2>
  <!-- [...] -->
</div>

<button
onclick="showLearningObjectives('ia2n')">Zeige Lernziele
</button>
```

Listing 2: HTML-ID und JavaScript mit Zufallsbezeichnern

enum-Aufzählungen mit fixen Werten zur Persistierung

Legacy-Code kann unter Umständen uralte oder selbstgeschriebene Persistierungsframeworks enthalten. Aufzählungen (klassischerweise, aber nicht zwingend `enum`s) haben für jedes Element einen fixen Wert hinterlegt, der für das Mapping von und zur Datenbank verwendet wird.

Listing 3 zeigt dies am Beispiel zu persistierender Farben. Ungünstig dabei ist zum einen, dass mit einer streng sequenziellen Ordnung `1`, `2`, `3`, `4` begonnen wurde. Dazu bestand überhaupt kein Grund; im Gegenteil, soll später neu die Farbe `ORANGE` hinzugefügt werden, dann muss sie wohl oder übel die Nummer `5` bekommen, obwohl sie besser zwischen `RED` und `YELLOW` gepasst hätte. Zum anderen können bei der Inspektion von Datenbankinhalten ähnliche Werte aus anderen Spalten (`Color` wird ja nicht die einzige derartige Spalte sein) die Übersicht und den Reverse-Lookup zum Quellcode erschweren.

```
public enum Color {
  RED(1),
  YELLOW(2),
  GREEN(3),
  BLUE(4);

  private final int value;

  private Color(int value) {
    this.value = value;
  }

  public int getValue() {
    return value;
  }
}
```

Listing 3: `enum`-Aufzählung mit ungünstigen sequenziellen Bezeichnern zu Persistierungszwecken

Hätte `Color.RED` hingegen den Wert `"WQWX"` (oder die Zufallszahl `8630`, wenn es unbedingt ein Integer sein muss), dann wäre ein Code-Lookup mittels Volltextsuche trivial. Einen umfassenden Blick hinter die Kulissen von `enum`s finden Sie übrigens in meinem JavaSPEKTRUM-Artikel [UYW7].

Anwendungsbeispiele in der Informatik allgemein

Nebst den bereits im Detail ausgeführten Shortlinks gibt es in der IT auch außerhalb der Softwareentwicklung diverse Anwendungsmöglichkeiten für Zufallsbezeichner:

- **Anker und Querverweise in großen Dokumenten** (insbesondere in LaTeX), zum Beispiel auf andere Abschnitte, Abbildungen oder Tabellen, auf Einträge im Literaturverzeichnis, aber auch Dateinamen für einzubindende Bilder lassen sich mit Vorteil mit zufälligen Bezeichnern versehen. Hier gilt wieder die gleiche Argumentation wie bei den oben beschriebenen HTML-IDs und JavaScript: Eine Systematik mit sprechenden Bezeichnern braucht extrem viel Disziplin, die nicht nur fehleranfällig ist, sondern sich unterm Strich wohl auch gar nicht lohnt.
- **Manuelle IP-Adressen und Port-Nummern im Heim- oder Firmennetzwerk** zeigen erfahrungsgemäß auch oft sequenzielle Zuweisungsreihenfolgen (oder Adressblöcke). Neue Geräte kommen hinzu und alte Geräte verschwinden. Letzten Endes muss eh in einer Tabelle nachgeschaut und diese ordentlich nachgeführt werden. Wieso also nicht gleich mit zufälligen internen IP-Adressen und Port-Freigaben beginnen?

Anwendungsbeispiele im Alltag

Auch im Alltag kann Zufall mit großem Vorteil eingesetzt werden. Sehr beeindruckend und ein Musterbeispiel hierfür ist die Lagerhaltung von Amazon. Der Onlineversandriese wendet eine sogenannte *chaotische Lagerhaltung* an. Dabei werden Artikel dort gelagert, wo es gerade Platz hat – unabhängig jeglicher Reihenfolgen, Produktkategorien oder dergleichen. Auf [QVAV] und [G399] sehen Sie in zwei Videos eindrücklich, wie das hinter den Kulissen von Amazon funktioniert.

Schülerinnen und Schüler wissen von den Vorteilen gemischter und damit zufälliger Lernkärtchen, vor allem beim Vokabellernen. Es versteht sich von selbst, dass das Lernen einer fixen Vokabelliste von

oben nach unten einen Bias in Richtung der oberen Wörter zur Folge hätte. Derartigen Bias versuche auch ich im Alltag zu vermeiden.

Ich bestimme das Musikalbum, das ich heute hören will, via Zufallsgenerator. Die Reihenfolge der Lieder ist selbstverständlich auch zufällig. Wenn ich mit dem Beantworten von aufgestauten (aber nicht dringenden) E-Mails nicht mehr nachkomme, lasse ich den Zufall entscheiden, mit welcher E-Mail ich mich als Nächstes befasse. Besteht ein Buch aus *unabhängigen* kleinen Kapiteln (zum Beispiel *„Effective Java"*), lasse ich den Zufall entscheiden, welches ich lese – sonst würde ich wohl immer nur die erste Hälfte des Buchs lesen. Erstelle ich Prüfungsfragen für meine Studierenden, verwende ich eine Liste aller prüfungsrelevanten Kapitel und lose dann eines aus, zu dem ich dann eine Aufgabe entwerfe.

Sie sehen, mir persönlich liegt viel daran, Bias in jedem Lebensbereich zu vermeiden – und ja, vielleicht bin ich auch ein bisschen verrückt … Machen wir also weiter mit etwas Mathematik.

Anzahl Variationen

Wie viele verschiedene Bezeichner lassen sich mit den beschriebenen Generatoren erzeugen? Betrachten wir der Einfachheit halber nur einen Viererblock aus Kleinbuchstaben, wie er zum Beispiel für meine eingangs erwähnten Shortlinks verwendet wird. Der Generatorzeichensatz besteht aus 10 Ziffern und 26 Kleinbuchstaben, abzüglich der zum Verwechseln ähnlichen Zeichen `1` und `l`. Das macht 34 verfügbare Zeichen, die im Urnenmodell *mit Zurücklegen* und *mit Berücksichtigung der Reihenfolge* gezogen werden. In der Mathematik spricht man in diesem Fall auch von *Variationen*. Pro Stelle existieren 34 Möglichkeiten, für vier Stellen also gesamthaft $34 \cdot 34 \cdot 34 \cdot 34 = 34^4 = 1\,336\,336$ Variationen. Das ist für diesen Anwendungszweck (Short-URLs) mehr als ausreichend.

Werden anstelle eines einzigen Viererblocks dessen zwei verwendet, dann existieren bereits $34^8 \approx 1{,}79 \cdot 10^{12}$, also 1,79 Billionen Möglichkeiten. Bei vier Viererblöcken sind es sogar $34^{16} \approx 3{,}19 \cdot 10^{24}$, also eine 25-stellige Anzahl möglicher Bezeichner. Tabelle 2 führt zu jeder Groß- und Kleinschreibung-Variante die Anzahl verfügbarer Zeichen und die sich daraus ergebende Anzahl Variationen auf.

Zum Vergleich: Rein zufallsbasierte UUIDs können $2^{122} \approx 5{,}32 \cdot 10^{36}$ verschiedene Werte repräsentieren, benötigen dafür aber auch brutto 36 Stellen Platz. „Kurzversionen" von UUIDs gibt es nicht. Die Groß- und Kleinbuchstaben-Variante meines Generators könnte auf diesen 36 Stellen Platz hingegen sieben Viererguppen (inklusive Bindestrich-Minus) und damit $1{,}19 \cdot 10^{47}$ verschiedene Bezeichner generieren, also 22 Milliarden mal mehr. Das veranschaulicht erneut, wie unergonomisch UUIDs mit ihrer hexadezimalen Darstellung sind.

Variante	Anzahl Zeichen	Anzahl Viererblöcke	Anzahl Variationen
nur Großbuchstaben	25	1	$390\,625 \approx 3{,}91 \cdot 10^5$
		2	$152\,587\,890\,625 \approx 1{,}53 \cdot 10^{11}$
		4	$23\,283\,064\,365\,386\,964\,000\,000 \approx 2{,}33 \cdot 10^{22}$
nur Kleinbuchstaben	34	1	$1\,336\,336 \approx 1{,}34 \cdot 10^6$
		2	$1\,785\,793\,904\,896 \approx 1{,}79 \cdot 10^{12}$
		4	$3\,189\,059\,870\,763\,704\,000\,000\,000 \approx 3{,}19 \cdot 10^{24}$
Groß- und Kleinbuchstaben	48	1	$5\,308\,416 \approx 5{,}31 \cdot 10^6$
		2	$28\,179\,280\,429\,056 \approx 2{,}82 \cdot 10^{13}$
		4	$794\,071\,845\,499\,378\,500\,000\,000\,000 \approx 7{,}94 \cdot 10^{26}$
Groß- oder Kleinbuchstaben	24	1	$331\,776 \approx 3{,}32 \cdot 10^5$
		2	$110\,075\,314\,176 \approx 1{,}10 \cdot 10^{11}$
		4	$12\,116\,574\,790\,945\,107\,000\,000 \approx 1{,}21 \cdot 10^{22}$

Tabelle 2: Anzahl Generatorzeichen und Variationen

Kollisionswahrscheinlichkeiten

Auch wenn wir jetzt zwar genau wissen, wie viele verschiedene IDs sich theoretisch generieren lassen, hilft uns diese Information nur bedingt weiter, da der Zufallsgenerator die bisher generierten Bezeichner nicht kennt und daher theoretisch eine bereits vorhandene Zeichenkette erneut generieren könnte. Die Frage ist also, wie groß die Wahrscheinlichkeit ist, dass von n generierten Bezeichnern mindestens zwei von ihnen den gleichen Wert haben.

Diese Fragestellung entspricht dem (zumindest unter Mathematikern) bekannten *Geburtstagsproblem* oder *Geburtstagsparadoxon* [J6QX]. Kennen Sie es schon? *Wie viele Personen müssen sich mindestens in einem Raum aufhalten, damit die Wahrscheinlichkeit, dass zwei oder mehr von ihnen am gleichen Datum (also Tag und Monat) Geburtstag haben, mehr als 50 Prozent beträgt?* Die Antwort wird Sie überraschen. Sie finden sie am Ende dieses Artikels unter [TUVQ].

Eine Verallgemeinerung des Geburtstagsproblems übertragen auf den Bereich der Kryptografie ist der sogenannte *Geburtstagsangriff* (englisch *birthday attack*). Die dahinterstehende Mathematik beant-

Abb. 1: Raumtemperaturfühler in einem großen Hallenbad mit vermeintlich „sprechenden" Bezeichnern

wortet nicht nur die Frage, wie hoch die Wahrscheinlichkeit für Kollisionen (in der Regel für Hashfunktionen) ist, sondern kann umgekehrt auch angeben, wie viele Werte *n* aus einer Wertemenge *N* generiert werden müssen, ehe sie mit einer Wahrscheinlichkeit *p* mindestens einen gleichen Wert enthalten [QQWV]. Die von mir als Service angebotene und oben bereits erwähnte Website [PKC4] zur Generierung von Zufallsbezeichnern enthält auch einen Rechner für die Anzahl generierbarer IDs für eine einstellbare Kollisionswahrscheinlichkeit.

Um Ihnen eine kurze Geschmacksprobe und Gespür für die Größenordnungen zu geben, betrachten wir zufällig generierte Primärschlüssel für eine Datenbanktabelle, bestehend aus zwei Viererblöcken mit Groß- und Kleinbuchstaben (zum Beispiel `soug-Ys9P`). Gemäß Tabelle 2 gibt es 28 Billionen verschiedene derartige Bezeichner. Wie viele Werte können wir vorab generieren und *in einem Schritt* in die Datenbank laden, damit die Wahrscheinlichkeit einer Kollision kleiner als 1 Promille ist?

Die Berechnung hierfür gibt ca. 237 000 Einträge. Anders ausgedrückt: Sie müssten im Mittel 1000 Datenbanktabellen mit jeweils 237 000 vorab zufällig generierten IDs erstellen, ehe es in einer Tabelle zu einer Kollision käme.

Bei vier Viererblöcken (zum Beispiel `Ggqc-T3XE-zVgi-GQ3q`) sind es bereits $1{,}26 \cdot 10^{12}$, also 1,26 Billionen mögliche IDs, „bis es chlöpft (knallt)" . Mit Sicherheit werden Sie nie so viele brauchen. Wenn Sie „nur" 40 Milliarden derartige IDs verwenden möchten, kommt es nur noch mit einer Wahrscheinlichkeit von 1 zu 1 Million zu einer Kollision. Können Sie sich mit 1 Million Tabelleneinträgen zufrieden geben, dann können Sie derartige IDs mit einer Sicherheit von 10^{15} zu 1 kollisionsfrei verwenden.

Fazit

Für das Kennzeichnen von Daten in der Informatik oder Gegenständen im Alltag gibt es diverse Systematiken:

- *Sequenzen:* beispielsweise 1, 2, 3, ...; 10, 20, 30, ..., A, B, C, ...
- *Wertebereiche:* beispielsweise Farben (rot, grün, blau, ...); Städte (München, Zürich, Hannover, ...); Planeten
- *„Sprechende" Bezeichner:* beispielsweise der Raumtemperaturfühler in Abbildung 1

In diesem Artikel habe ich Ihnen neu die Vorteile und Hintergründe von *Zufallsbezeichnern* aufgezeigt und Ihnen gleich dazu die passenden Handwerkzeuge mitgegeben. Sollten Sie in Zukunft wieder einmal vor der Aufgabe stehen, passende Bezeichner für eine „Datenmenge" irgendeiner Art zu finden, dann empfehle ich Ihnen, sich selbst vorab folgende Fragen zu stellen:

- Haben die einzelnen Objekte eine natürliche Ordnung oder inhärente Reihenfolge, die es wirklich rechtfertigt, sie sequenziell zu kennzeichnen?
- Kann garantiert ausgeschlossen werden, dass die Reihenfolge eines Tages nicht mehr eingehalten werden kann, vorreservierte Blöcke überfüllen oder Lücken entstehen?
- Hat die Systematik vermeintlich sprechender Bezeichner wirklich einen Mehrwert, oder werden sie letzten Endes doch wieder in einer Tabelle nachgeschaut und gepflegt werden müssen?
- Geben die Bezeichner mit Sicherheit keine unfreiwilligen Informationen nach außen preis, weder direkt noch indirekt?

Können für den angedachten Typ nicht alle Fragen mit einem überzeugenden „Ja" beantwortet werden, bieten die hier vorgestellten Zufallsbezeichner eine anfangs vielleicht ungewohnte, aber mit Sicherheit überzeugende Alternative. Denn es wäre ausgesprochen ärgerlich, wenn eine neu eingeführte Systematik mit der Zeit in sich zusammenfällt, aber nachträglich auch nicht mehr geändert werden kann.

Ich freue mich, von Ihren neuen Informatik- und Alltagserfahrungen mit Zufallsbezeichnern zu lesen. Sie erreichen mich unter [MWJG] ;-)

Literatur, Links und Noten

[3LPH] Java Platform, Standard Edition & Java Development Kit Version 15 API Specification, Class UUID, https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/UUID.html
[3YKU] C. Heitzmann, Theorie und Praxis von Zufallszahlengeneratoren, in: JavaSPEKTRUM, 2/2019
[AAPH] G. A. Miller, The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information, in: Psychological Review 63, 1956
[EXFQ] Internet Engineering Task Force (IETF), RFC 4122, A Universally Unique IDentifier (UUID) URN Namespace, https://www.rfc-editor.org/info/rfc4122
[FVEK] Quellcode zum Herunterladen, https://link.simplexacode.ch/x48n
[G399] YouTube, Inside Amazon Teil 2: So funktioniert ein Amazon Logistikzentrum!, https://www.youtube.com/watch?v=GzxyY4NxeWg
[HE4L] C. Heitzmann, Future und CompletableFuture, in: JavaSPEKTRUM, 2/2021
[J6QX] Wikipedia, Geburtstagsparadoxon, https://de.wikipedia.org/wiki/Geburtstagsparadoxon
[MWJG] christian.heitzmann@simplexacode.ch
[PKC4] SimplexaCode, ID Generator, https://link.simplexacode.ch/bk4k
[PNQF] D. J. Levitin, The Organized Mind: Thinking Straight in the Age of Information Overload, Chapter 2: The First Things to Get Straight: How Attention and Memory Work, Dutton, 2014
[QQWV] Wikipedia, Birthday attack, https://en.wikipedia.org/wiki/Birthday_attack
[QVAV] YouTube, Inside Amazon: So funktioniert ein Amazon Logistikzentrum! - Teil 1, https://www.youtube.com/watch?v=hiZIk45tga8
[TUVQ] mindestens 23 Personen
[UYW7] C. Heitzmann, Der Einsatz von enums abseits reiner Aufzählungen, in: JavaSPEKTRUM, 1/2019
[W6A6] RANDOM.ORG, https://www.random.org