www.javaspektrum.de

Ausgabe 5/2019
D € 9,90 · A € 10,90 · CHF 18,20

4 194156 109901 05

JavaSPEKTRUM

Magazin für professionelle Entwicklung und Integration von Enterprise-Systemen

Java @ IoT –
Anwendungen für das Internet der Dinge

Baukasten der Dinge – IoT in der Cloud

Java im IoT nutzen, wo es die beste Alternative ist

Interview

Microsoft Deutschland Chefin Sabine Bendiek über KI-Strat… die Notwendigkeit zur W… und die Demokrati…

…s Monolith – …vert von Microservices

…rity @ Kubernetes Network Policies Teil 1: Good Practices

Sonde… …ere Magazin – Jobs in der Softwarebranche

Java @ IoT – Anwendungen für das Internet der Dinge

G30325

Write once, run on all IoT devices

Java im IoT nutzen, wo es die beste Alternative ist!

Marvin Wiegand, Jonas Grundler

IoT ist seit Langem *der* Megatrend schlechthin. Plattformen, Geräte, Protokolle, Architekturen, weitere Trends wie Künstliche Intelligenz (KI) und Distributed-Ledger-Technik – alles, was benötigt wird, um IoT-Szenarien zu ermöglichen, wird uns von der Community und der Wirtschaft zur Verfügung gestellt. Dabei ist es nicht einfach, einen Überblick zu gewinnen. Dieser Artikel bietet einen einfachen Einstieg in das Thema anhand einer Demo, die sich mit verschiedenen (IoT-)Fragestellungen auseinandersetzt. Im Sinne von „Write once, run on all IoT devices" betrachten wir dabei, wo Java seinen Platz einnimmt.

Stellen Sie sich vor, Sie könnten Daten an Orten abgreifen, an denen Sie sich nicht befinden – zuverlässig und sicher. So sicher und nachweisbar, dass Sie damit eine automatische Zahlung veranlassen können. Stellen Sie sich vor, Sie wären Parkhausbetreiber – und wollten ticketloses Einfahren ermöglichen – ebenso das Laden von Elektroautos, ohne menschliches Interagieren. Das Szenario ist schnell durchdacht: Schranken, Ladestationen, sichere Kommunikation und Einbindung eines Bezahldienstes.

Aber welche Komponenten und welche Techniken werden nach Stand der Technik eingesetzt? Und wo bietet Java seinen Mehrwert? Lesen Sie quer – lesen Sie im Detail – und machen Sie sich Ihr persönliches Bild der Möglichkeiten!

Szenario

Das diesem Artikel zugrunde liegende Szenario lässt sich mit wenigen Worten gut beschreiben: Ein Parkhaus (PH) darf von einem Fahrzeug (FZ) nur dann befahren werden (Schranke), wenn es den Vertragsbedingungen zugestimmt hat. Elektrofahrzeuge können auf entsprechend ausgestatteten Parkplätzen Strom tanken.

Die Abrechnung gestaltet sich folgendermaßen: Es wird Miete für den Parkplatz berechnet, außerdem können Kosten für das (Strom-)Tanken entstehen. Zudem fallen Kosten an, wenn eine Ladestation besetzt und kein Strom getankt wird, da dann die Ladeinfrastruktur blockiert ist und keine Umsätze entstehen. Weiter soll die Abrechnung zeitgemäß möglich sein. Bei der Umsetzung und im weiteren Verlauf dieses Artikels betrachten wir Folgendes: Nachvollziehbarkeit (von Zahlungen, Ladevorgängen, Parkvorgängen), Over-the-Air Updates und Kommunikation zwischen den Komponenten. Auch Java hat in dieser Welt seinen Platz – wo, zeigen wir besonders gerne.

Um das Szenario im ersten Ansatz genauer zu verstehen, blicken wir auf die konzeptionelle Darstellung in Abbildung 1. Dort entdecken wir einen Broker, der für den Datenaustausch zwischen den FZ, dem PH und der Infrastruktur verantwortlich ist. Außerdem gibt es einen Distributed Ledger (DL, [DLT]), der die drei Teilnehmer FZ, Stromanbieter und PH miteinander verknüpft. Hier findet offenbar gesicherter und nicht veränderbarer Datenaustausch statt. Außerdem wird über den DL die Bezahlung organisiert.

Um es vorwegzunehmen: Java erleichtert die Entwicklung solcher Szenarien entscheidend. Später im Text werden wir genauer verstehen, warum. Und für alle Interessierten: Der Quellcode, Anleitungen usw. für diese Demo ist unter [SRC] zu finden.

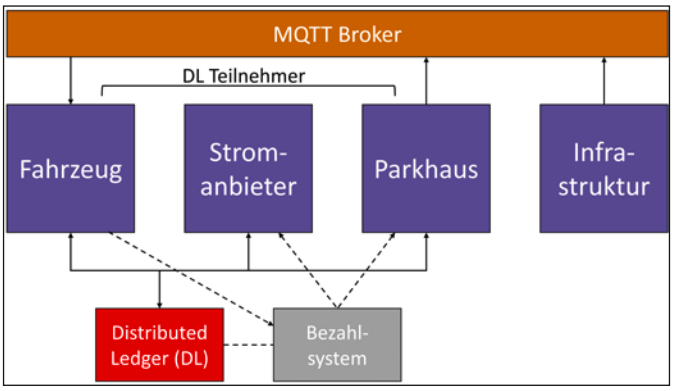

Abb. 1: Szenario: Konzeptionelle Darstellung

Marvin Wiegand arbeitet im Bereich Innovation bei der Novatec Consulting GmbH. Dort befasst er sich insbesondere mit Themen wie Cloud, Machine Learning, IoT und Distributed Ledger. Das Zusammenspiel dieser Technologien begeistert ihn! Gemeinsam mit Kunden testet er sie, implementiert Prototypen und verknüpft sie zu neuen und spannenden Lösungswegen. E-Mail: marvin.wiegand@novatec-gmbh.de

Jonas Grundler ist Head of Digital Innovation bei der Novatec Consulting GmbH. Er hat in seiner Rolle unterschiedliche Themen aufgebaut – angefangen bei Cloud bis hin zu Machine Learning. Seit einiger Zeit konzentriert er sich auf IoT, ein Themengebiet, in dem er seine Leidenschaft für Neues und kreative Lösungsfindung ausleben und genießen kann. E-Mail: jonas.grundler@novatec-gmbh.de

Infrastruktur

Die Infrastruktur ist auf einem quadratischen Holzbrett von 80 cm x 80 cm untergebracht. Dort befinden sich alle Elemente, die für das FZ und die Szenarien notwendig sind:

Die Verkehrsführung ist über eine *schwarze Linie* (s. Abb. 2) realisiert, anhand derer sich die FZ orientieren (Stichwort Line Follower). Es sind zwei ineinandergreifende Schleifen untergebracht: Eine Schleife, die am Parkhaus vorbeiführt – falls das FZ mit den Vertragsbedingungen nicht einverstanden ist. Und eine zweite Schleife, die in das PH hineinführt – falls das FZ parken und gegebenenfalls Strom tanken möchte.

Über *schwarze Balken* quer zur Fahrtrichtung in Verbindung mit passend platzierten NFC-Sensoren [NFC] erkennt das FZ, dass es anhalten soll. Die Infrastruktur kann dann über den dort platzierten NFC-Sensor das FZ identifizieren und es mit Informationen versorgen. Das FZ kann daraufhin weitere Entscheidungen (bspw. Einfahrt in PH ja/nein, Einfahrt in Parkplatz ja/nein) treffen. Eine *Schranke* an der Einfahrt zum PH öffnet nur dann, wenn das FZ einfahren darf. Eine Schranke an der Ausfahrt vom PH öffnet nur dann, wenn das FZ die angefallenen Kosten beglichen hat.

Zusätzlich zur Infrastruktur für die direkte Interaktion mit dem FZ (NFC-Sensoren, Schranken, Linien und Balken) ist ein Raspberry Pi [RPI] verbaut, der die Steuerung der Sensoren und Aktoren übernimmt. Außerdem stellt der RPI eine Verbindung zu Diensten in der Cloud her. An dieser Stelle soll nicht unerwähnt bleiben, dass die Infrastruktur wie vorgestellt nur eine von vielen Möglichkeiten ist.

NFC-Sensoren

Das Brett besitzt also schwarze (Quer-)Balken und *NFC-Sensoren*. Das FZ hat an der Unterseite ein RFID-Tag [RFID] verbaut, über welches es identifiziert wird beziehungsweise sich ausweisen kann. Wir haben diese Kombination für folgende Situationen verbaut:

- *FZ steht vor der Einfahrt zum PH:* Kommt es zum Vertrag zwischen FZ und PH-Betreiber, öffnet sich die Schranke und das FZ darf einfahren.
- *FZ steht vor einem Parkplatz* und möchte wissen, ob es einfahren darf oder nicht. Diese Information bekommt es von der Infrastruktur mitgeteilt.
- *FZ steht auf einem Parkplatz:* Über den NFC-Sensor auf dem Parkplatz weiß die Infrastruktur, ob und von welchem FZ der Parkplatz genutzt wird.
- *FZ möchte ausfahren:* Die Schranke öffnet sich erst, nachdem der verbrauchte Betrag überwiesen wurde.
- *FZ steht an einer Kreuzung* und möchte diese überqueren oder einmünden.

Qi-Ladestation

Manche Parkplätze sind mit einer Ladestation über den Qi-Standard [QI] ausgestattet. Die Betreiber (es kann unterschiedliche geben) der Ladestationen sind im DL-Netzwerk bekannt. Bei Einfahrt in das PH werden die Vertragsbedingungen mitgeteilt.

Kommunikationswege

Die NFC-Sensoren und Schranken werden über den RPI über I²C [I2C] angesteuert. Der RPI dient als Edge-Device für die Infrastruktur und stellt Verbindungen zu Cloud-Diensten her, beispielsweise zur Kommunikation mit dem FZ oder dem Abrech-

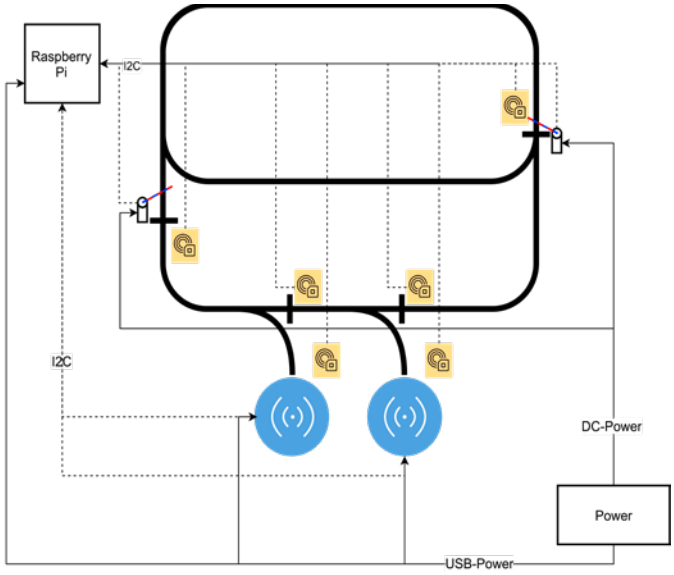

Abb. 2: Infrastruktur: Schematische Darstellung

Abb. 3: Fahrzeug: Frisch gedruckt

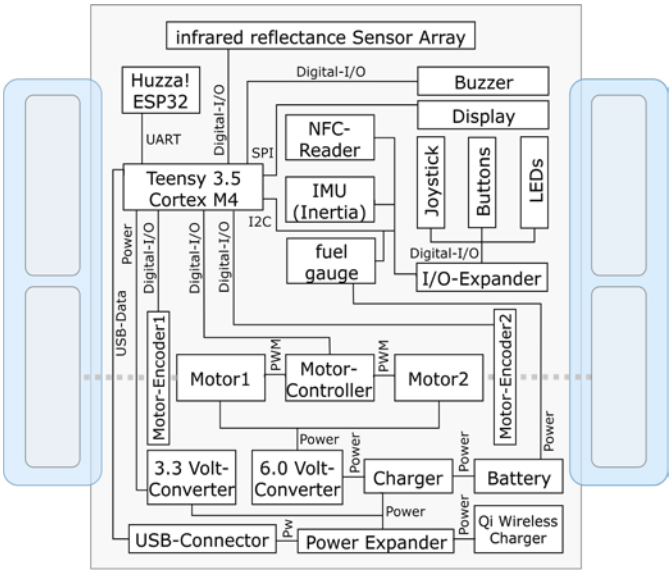

Abb. 4: Fahrzeug: Schematische Darstellung der Hardware

nungsservice des PH-Betreibers. Das FZ stellt ebenso eine (eigene) Verbindung zu Cloud-Diensten her – so kann es mit der Infrastruktur, dem PH-Betreiber und dem Stromanbieter kommunizieren.

Fahrzeug

Nachdem die Infrastruktur aufgebaut und verstanden wurde, wenden wir uns nun der mobilen Komponente zu. Wir sprechen von einem Fahrzeug (FZ, s. Abb. 3), welches sich anhand der Linien und Sensoren/Balken-Kombinationen auf dem Brett eigenständig bewegen kann.

Zur Linienverfolgung werden die Helligkeitswerte, die das Array aus sechs digitalen Infrarotsensoren aufnimmt, an den Cortex-M4-Prozessor ([CM4], s. Abb. 4) übertragen. Dieser ermittelt mithilfe eines PID-Controllers [PID] den Änderungsbedarf, der an die Motorsteuerung weitergegeben werden muss, damit das FZ weiterhin der vorgegebenen Linie folgt. Fährt das FZ nun auf einen schwarzen Querbalken, so hält das FZ an und wartet auf Daten, die es über ein FZ-spezifisches MQTT-Topic [MQTT] empfängt. Die Infrastruktur stellt also über den entsprechend positionierten NFC-Sensor fest, dass sich dort ein FZ in Warteposition befindet.

Entsprechend kann sie nun:

- ein Angebot dem FZ unterbreiten (Einfahrt in das PH),
- Guthaben zum Transfer anweisen (Ausfahrt),
- mitteilen, ob der nächste Parkplatz belegt ist – oder an Kreuzungen entsprechend zur Weiterfahrt auffordern oder weiteres Warten motivieren.

Für die WLAN-Verbindung des FZ ist der ESP32 [ESP32] zuständig. Es werden verschiedene Protokolle verwendet: MQTT und gRPC [GRPC] (im Falle von DL). Außerdem findet ein Austausch von Programmcode über die ArduinoOTA-Bibliothek [AOTA] statt (s. Listing 1).

Steht das FZ auf einem Parkplatz mit Ladestation, kann es den Betreiber der Ladestation auffordern, Strom zu liefern. Die Voraussetzungen hierzu wurden bei der Erstellung des Vertrags vor Einfahrt in das PH gelegt. Dazu meldet das Fahrzeug seinen Wunsch per Hyperledger Fabric (HLF)-Client an das DL-Netzwerk. Der Chaincode des DL übernimmt nun die Kommunikation mit dem Betreiber der Ladestation.

```
// C++
// motor control based on infrared sensor array
...
// read sensor values (0-5000) and
// obtain line deviation from center of sensor array
deviation = qtr.readLineBlack(sensorValues) - 2500;

// pid uses variable deviation,
// tries to achieve an lineDev/outputVal of zero
pid.run();

// adjust motors
speedLeft = baseSpeed + outputVal;
speedRight = baseSpeed - outputVal;

motors.setSpeed(speedLeft, speedRight);
```

Listing 1: PID-Controller auf Cortex-M4: Ausschnitt Steuerung

Zusätzlich zu dem für den Betrieb notwendigen Informationsaustausch versendet das FZ aktuelle Sensordaten mit aufbereitetem Kontext per MQTT an die Cloud. Der Kontext enthält Daten über den Ladezustand der Batterie, einen Zeitstempel und ob sich das FZ im Einpark-, Park-, Fahr- oder Warte-Modus befindet. Die Daten stehen nun Analysewerkzeugen zur Verfügung, um beispielsweise den PID-Controller zu justieren, Parkvorgänge genauer und schneller zu gestalten oder ganz allgemein Probleme und deren Ursachen zu erkennen. Hier bieten sich auch Machine Learning-Methoden an. Mittels Over-the-Air Updates kann optimierte oder angepasste Firmware auf das FZ beziehungsweise dessen Prozessoren (Cortex-M4 oder ESP32) aufgespielt werden.

Tabelle 1 listet die relevantesten Komponenten und deren Funktionen unseres Szenarios auf.

In Abbildung 3 ist das FZ schematisch dargestellt. Links und rechts sind insgesamt vier Räder, die die Kette führen. Neben den Bezeichnungen der Komponenten sind in Abbildung 4 deren Verbindungen in Form eines einfachen Schaltplans sichtbar.

Die Programmierung der Prozessoren (ESP32 und Cortex-M4) wurde in C/C++ vorgenommen. Warum nicht in Java? Google spricht hier eine eindeutige Sprache: Es gab immer wieder Versuche, Ja-

Komponente	Funktion
Array aus sechs digitalen Infrarotsensoren	Linien/Balkenerkennung im Infrarotspektrum (unempfindlicher gegen Helligkeitsschwankungen als der für uns sichtbare Bereich)
ESP32	drahtlose Kommunikation und über UART mit Cortex-M4 verbunden
Cortex-M4	Motorsteuerung mithilfe eines PID-Controllers (Eingabe ist das Array aus sechs digitalen Infrarotsensoren)
Akku mit Batterie-Babysitter	Über den Qi-Standard ladbare Lithium-Ionen-Batterie
LED-Display mit Joystick & Buttons	manuelle Konfiguration des Cortex-M4, außerdem Steuerung der Motoren zu Testzwecken
MQTT-Client	HiveMQ [HMQ], zur Übermittlung von Sensordaten in die IoT-Plattform; außerdem Empfang von Daten - bspw. „Parkplatz belegt“
Distributed-Ledger-Client	Hyperledger Fabric, per gRPC wird mit dem DL-Netzwerk kommuniziert

Tabelle 1: Komponenten und deren Funktionen unseres Szenarios

va auf (low-power) Mikrocontroller zu portieren. Auch Java ME ist häufig im Gespräch. Jedoch hat sich in keiner Mikrocontroller-Community, die unser Szenario betrifft, Java durchgesetzt. Wir stoßen hauptsächlich auf C/C++, Lua [LUA] und spezialisierte Sprachen. Entsprechend groß wäre der Aufwand, Java einzusetzen – nützliche Beispiele müssten übersetzt und eigene Bibliotheken erstellt werden. Ein großer Mehrwert von Java, unbeschwert zwischen JVM-Installationen Quelltext auszutauschen, wäre dahin.

Raspberry Pi

Für die Steuerung der Infrastruktur nutzen wir einen Raspberry Pi (RPI). Das hat den Vorteil, dass wir eine Hochsprache wie Java nutzen können und uns dabei in bester Gesellschaft wiederfinden. Zudem können wir von vielen weiteren Annehmlichkeiten profitieren. Der RPI nimmt drei Rollen ein:

- *Infrastruktur:* Kommunikation mit den FZ und Abfrage der NFC-Sensoren außerhalb des Parkhauses
- *PH:* Kommunikation mit DL als Teilnehmer, Kommunikation mit den FZ, Abfrage der NFC-Sensoren innerhalb des PH, Steuerung der Schranken
- *Stromanbieter:* Kommunikation mit DL als Teilnehmer, Stromlieferung, Berechnung Stromverbrauch

Zur Kommunikation mit den FZ wird die HiveMQ MQTT Client Java library verwendet. Über diesen Client werden Nachrichten zu den FZ beziehungsweise an FZ-spezifische Topics versendet. Als Name des Topics eignet sich beispielsweise die RFID-Kennung des FZ als Subtopic der Kategorie Ort und car, beispielsweise infrastructure/car/{rfid} oder carpark/{carpark-id}/car/{rfid}. Als Servicelevel schlagen wir *at most once* vor, da dies die (ressourcen-) günstigste Variante ist. Nur die Infrastruktur und das PH dürfen Nachrichten in die Topics car/{rfid} publizieren. Die FZ dürfen jeweils nur Topics abonnieren, die ihre eigene RFID-Kennung enthalten (die RFID-Kennung ist einem FZ über eine Umgebungsvariable bekannt).

Sobald sich ein FZ über einem NFC-Sensor befindet, steht die RFID-Kennung der Infrastruktur bereit. Der RPI liest periodisch die Daten aller NFC-Sensoren und weiß damit, welche FZ auf Information warten. Da der RPI das Layout des Bretts kennt, kann er sensorspezifisch Daten den FZ mitteilen. Jedes Auslesen einer RFID-Kennung führt zum Versand einer spezifischen Nachricht – dieses Vorgehen garantiert, dass die FZ die benötigte Information bekommen. Ein zusätzlicher Vorteil beispielsweise gegenüber einem einmaligen Versand der Information ist, dass weder FZ noch RPI einen Zustand abspeichern müssen. Nicht zu vergessen ist jedoch, dass das FZ wissen muss, wo es sich befindet (hier: Infrastruktur oder PH), um auf das richtige Topic zu hören.

```
...
// Java
// Sample code demonstrating how events like a contract
creation
// can be received and reacted upon
import org.hyperledger.fabric.sdk.*;
...
ChaincodeEventListener cel = new ChaincodeEventListener() {
  @Override
  public void received(
     String handle, BlockEvent be, ChaincodeEvent ce) {
    // write code here to open the parking barrier
    // and inform the car
    // ...
};
...
channel.registerChaincodeEventListener(
  Pattern.compile(".*"),
  Pattern.compile(".*")),
   cel);
```

Listing 2: Hyperledger Fabric Client: Empfangen von API Events

Steht ein FZ vor der Einfahrt in das PH, wird dem FZ zunächst per MQTT die ID des PH mitgeteilt – damit ist das MQTT-Topic festgelegt, welches für die Navigation innerhalb des PH verwendet wird. Dann wird der Vertrag ausgehandelt. Im Erfolgsfall bekommt der auf dem RPI laufende HLF-Client die Information, dass ein Vertrag zustande gekommen ist (s. Listing 2). Der RPI kann dann die Schranke öffnen (über das MQTT-Topic für die Kommunikation zwischen PH und FZ). Die Schranke schließt der Einfachheit halber nach 15 Sekunden.

Ob ein Parkplatz angefahren werden kann oder nicht, teilt der RPI dem Fahrzeug an den entsprechenden Stellen auf dem Brett mit. Steht ein FZ auf einem Parkplatz mit Ladestation, kann es über den HLF-Java-Client die Stromlieferung beantragen. Die Anfrage nimmt der RPI in seiner Rolle als Stromanbieter entgegen und startet den Stromfluss an der Ladestation. Gleichzeitig wird der Start der Nutzung im DL protokolliert. Der Verbrauch wird gesetzeskonform aufgezeichnet beziehungsweise im Rahmen des Szenarios nachempfunden.

Steht ein FZ vor der Ausfahrt aus dem PH, teilt der RPI dem FZ mit, dass es nun die angefallenen Kosten begleichen soll. Das FZ weist den DL an, die Beträge unter Zuhilfenahme des Bezahldienstes zu transferieren. Daraufhin wird der RPI über das Ende des Vertrags vom DL informiert und öffnet die Schranke. Das FZ bekommt die Information, dass die Schranke geöffnet ist, und kann ausfahren.

Die Programmierung des RPI ist in Java vorgenommen. Es gibt andere Sprachen, die sich ebenfalls anbieten. Die von uns eingesetzten Technologien (Webkomponenten, HLF, HiveMQ-Client, Schnittstellen für GPIO [GPIO]) sind hervorragend in Java unterstützt. Als Java-Fan gibt es keinen Grund, von seiner Leidenschaft abzuweichen!

Cloud

In der Cloud sind zwei Dienste abgebildet: MQTT-Broker, welcher die Kommunikation zwischen FZ und Infrastruktur und PH sicherstellt, und außerdem ein DL, der die Freigaben, Verträge und Nutzungen regelt.

MQTT-Broker

Hierfür verwenden wir den Broker HiveMQ, der durch seine Skalierbarkeit und anwenderfreundliche Bedienung besticht. Außerdem steht ein Java-Client zur Verfügung. Die Konfiguration unseres HiveMQ-Brokers erzwingt die Authentifizierung von Clients, da diesen unterschiedliche Autorisierungen zugeordnet werden müssen:

Die *Infrastruktur* darf auf infrastructure/car/# Nachrichten veröffentlichen. Das ist in unserem Szenario immer dann der Fall, wenn sich das FZ nicht im PH befindet. Das *PH* darf auf carpark/{carpark-id}/car/# Nachrichten veröffentlichen. Die carpark-id ist statisch; in unserem Fall verwenden wir Parkhaus-42. Das *FZ* darf infrastructure/car/{rfid} und carpark/+/car/{rfid} abonnieren.

Die MQTT-Clients müssen vor Nutzung des HiveMQ-Brokers registriert werden. In diesem Zuge erhält der Nutzer Zugriff auf die ihm zugeordneten Topics.

Distributed Ledger

In unserem Szenario kommt HLF zum Einsatz. Damit folgen wir einer großen Community unter dem Dach der Linux Foundation [LF]. HLF ist ein Framework für die Verteilung verifizierter Information. Updates dieser Information werden mit Prüfsummen verknüpft – somit können historische Daten nicht unentdeckt manipuliert werden.

Damit FZ (und weitere) an Transaktionen teilnehmen können, müssen sie sich zuvor registrieren (permissioned DLT [PDLT]). Dafür benötigen sie bei HLF eine Certificate Authority [CA], die einen Membership Service Provider (MSP) für sie bereithält. Bei diesem MSP können sich Benutzer (FZ) anmelden. Außerdem wird in unserem Szenario verlangt, dass ein Bezahldienst konfiguriert und im Account hinterlegt wird.

Der Einsatz von HiveMQ und Hyperledger Fabric erfordert zunächst keine Java-Kenntnisse. Will man Spezialanforderungen umsetzen, wie Client-spezifische Topics, so geht das aber insbesondere einfach mithilfe eines Java-Plug-ins für HiveMQ. Eine Weiterverarbeitung von MQTT-Nachrichten, die Darstellung des Parkhauses in der digitalen Welt oder Steuerungsmöglichkeiten lassen sich gemäß moderner Entwicklung für die Cloud umsetzen. Dabei spielt Java oft eine große Rolle, je nach Anwendungsfall aber auch andere Sprachen: Python oder R [R] (Machine Learning), Kotlin [KTL] oder auch Typescript. Meist ist es eindeutig, wann der Java-Entwickler in seiner Welt bleiben kann, und wann es geschickter ist, eine andere Sprache zu wählen.

Distributed-Ledger-Konfiguration

Dieses Szenario besitzt *einen* (HLF) Channel. Verschiedene Teilnehmer beziehungsweise Teilnehmergruppen (Infrastruktur, PH-Betreiber, Stromanbieter mit Ladestationen, FZ(e)) können über den Channel Daten austauschen, beispielsweise um Nutzungsvereinbarungen zu treffen und Verträge abzuwickeln. Hierfür gibt es zwei definierte Assets (s. Listing 3):

- *Vertrag:* Bei der Einfahrt in das PH sind hier die Bedingungen festlegt – beispielsweise Parkgebühren der Parkplätze, außerdem die Kosten und Ansprechpartner der Ladestationen. Das Datenmodell erlaubt die Modellierung unterschiedlicher Verträge – zum Beispiel für Dauerparker, Anlieger mit speziellen Konditionen, Spontanparker usw. Bei Einfahrt stehen gegebenenfalls mehrere Vertragsvarianten zur Verfügung. In der von uns abgebildeten Demo wird nur eine Vertragsart zur Verfügung gestellt.

```
// Hyperledger Composer Model Language
// define assets Parking and Contract
namespace novatec.parking.charging
...
asset Parking identified by parkingId {
  o String parkingId
  ...
  o Double batteryCharge optional
  --> Contract contract
}
...
asset Contract identified by contractId {
  o String contractId
  --> ParkingSpotProvider parkingSpotProvider
  --> EnergyProvider energyProvider
  --> Vehicle vehicle
  o DateTime arrivalDateTime
  o Double chargingPrice
  o ...
}
```

Listing 3: Distributed Ledger: Ausschnitt Datenmodell – Assets-Nutzung (Parking) und Vertrag (Contract)

- *Nutzung:* Bei der Einfahrt in das PH wird eine Nutzung zu den festgelegten Konditionen angelegt. Zunächst wird hierbei nur die Parkplatzmiete aufgezeichnet, sobald sich das FZ auf einem Parkplatz befindet. Möglich ist, dass im späteren Verlauf das Laden an einer Ladesäule auf einem Parkplatz zu den Konditionen des Stromanbieters dieser Ladesäule hinzukommt.

Zu den definierten Assets sollten wir noch die Verwaltung der Accounts genauer betrachten: Der DL hat mittels eines Bezahldienstes die Möglichkeit, Rechnungen zu stellen beziehungsweise Guthaben zum Transfer anzuweisen. Dadurch ist sichergestellt, dass Parkplatzmieten und Stromgebühren bezahlt werden können. Im Streitfall kann nachgewiesen werden, ob und wann welche Kosten entstanden sind.

Fazit

Wer sich mit Java dem Megatrend IoT nähern möchte, wird zunächst von der Vielfalt bereits existierender Projekte überrascht. Java scheint sämtliche Bereiche durchdrungen zu haben – von Mikrocontrollern bis hin in die Cloud. Bei näherem Hinsehen kristallisiert sich jedoch aus unserer Sicht heraus, dass Java genau dann eingesetzt werden sollte, wenn es seine Stärken ausspielen kann: Integration verschiedener Technologien, Nutzung komplexer Systeme, Skalierbarkeit, Flexibilität. Der Trend aus der Microservices-Welt zur polyglotten Entwicklung sollte zumindest zum Nachdenken anregen. In diesem Sinne – Java nutzen dort, wo es die beste Alternative ist!

PS: Einen besonderen Dank an Titus Meyer (Titus.Meyer@novatec-gmbh.de) dürfen wir nicht vergessen – ohne seine Mithilfe, seine Erfahrung und seinen 3D-Drucker wäre das Projekt nicht möglich gewesen!

Links

[AOTA] https://github.com/esp8266/Arduino/tree/master/libraries/ArduinoOTA
[CA] https://de.wikipedia.org/wiki/Zertifizierungsstelle
[CM4] https://de.wikipedia.org/wiki/ARM_Cortex-M4
[DLT] https://de.wikipedia.org/wiki/Distributed-Ledger-Technologie
[ESP32] http://esp32.net/
[GPIO] https://www.raspberrypi.org/documentation/usage/gpio/
[GRPC] https://grpc.io/
[HLF] https://www.hyperledger.org/projects/fabric
[HMQ] https://www.hivemq.com/
[I2C] https://www.nxp.com/docs/en/user-guide/UM10204.pdf
[KTL] https://kotlinlang.org/
[LF] https://www.linuxfoundation.org/
[LUA] https://www.lua.org/
[MQTT] http://mqtt.org/
[NFC] https://de.wikipedia.org/wiki/Near_Field_Communication
[PDLT] https://wirtschaftslexikon.gabler.de/definition/distributed-ledger-technologie-dlt-54410
[PID] https://en.wikipedia.org/wiki/PID_controller
[QI] https://www.wirelesspowerconsortium.com/qi/
[R] https://www.r-project.org/
[RFID] https://de.wikipedia.org/wiki/RFID
[RPI] https://www.raspberrypi.org/
[SRC] https://github.com/NovatecConsulting/carpark-demo