Test Szenario

Aufgabenstellung

Ziel der Aufgabe war es, einen Test Fuhrpark auf dem Hyperledger Service in der SCP zu kreieren - noch ohne echten Daten vorläufig

Stand des Test Fuhrparks

Der Fuhrpark beinhaltet nun die beiden Objekte - **User** & **Car**. Man ist nun in der Lage User und Autos mit einer definierten Struktur zu erstellen und abzufragen. Ebenfalls lassen sich diese Objekte verändern (updaten) und löschen.

Als User ist man in der Lage sich einen Auto auszuleihen, sofern der User noch kein Auto ausgeliehen hat und das Auto ebenfalls noch nicht belegt ist. Es wird automatisch dazu ein dazugehöriges Ausleihen Objekt erstellt, welches man abfragen kann.

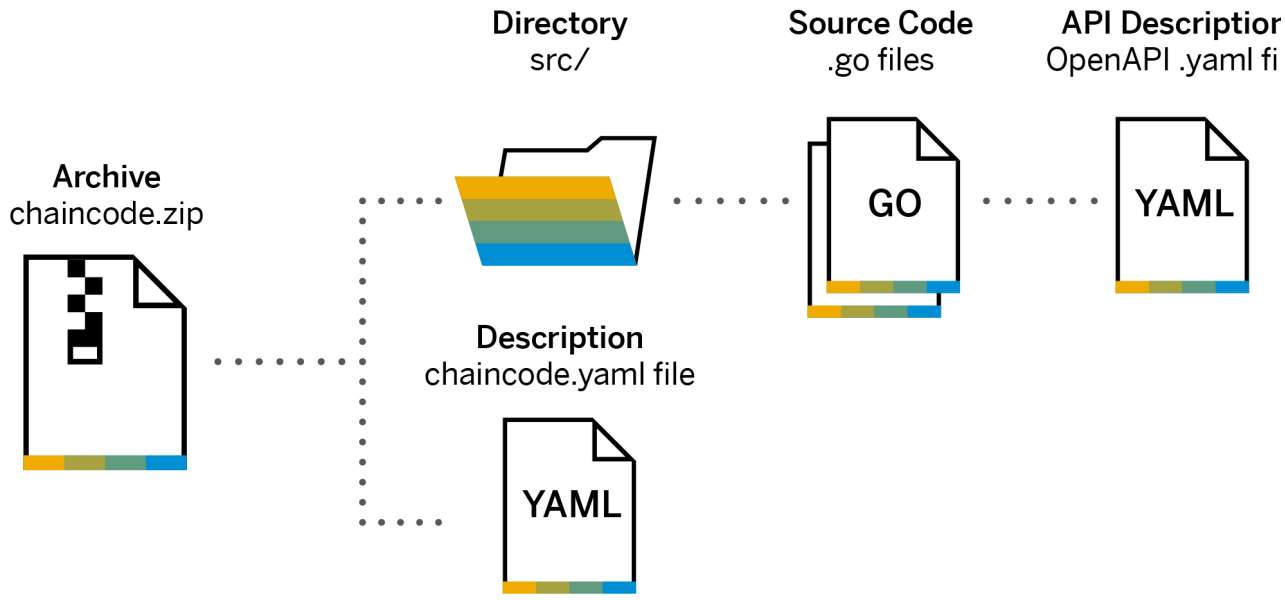
Der User kann dieses Auto nun wieder abgeben und beide Objekte sind nun wieder als nicht belegt markiert. Ein Fahrtenbuch wird voll automatisch dazu erstellt, welches auch abgefragt werden kann.

Wie funktioniert das ganze?

Aufbau des kompletten Programms[1]

Zwar kann man Hyperledger Programme auch in JavaScript schreiben, der SCP Service akzeptiert jedoch (vorerst) nur Programme in der Programmiersprache GO. Dieses Programm wird bei Hyperledger Chaincode genannt (bei vielen anderen Blockchain Projekten heißt das Smart Contract). In diesem Chaincode ist die komplette Business Logik (Geschäftslogik) implementiert. Die Schnittstelle zu diesem Chaincode stellt eine REST API dar, welche im .yaml Format geschrieben werden muss. Beide Dateien müssen sich im Unterordner src/ befinden.

Copyright © 2017, Fink IT-Solutions GmbH & Co.KG.



Neben dem src Ordner, in welchem sich eigentlich alles befindet, gibt es noch eine einzige "Description" Datei welche chaincode.yaml heißen muss. In dieser steht lediglich eine Beschreibung, der Name des kompletten Programms und die Version.

Der src Ordner muss nun gemeinsam mit dem chaincode.yaml "gezipped" werden und diese Datei kann dann in der SCP hochgeladen und installiert werden. Dafür muss man in seine eigene Node gehen Chaincode + Install Chaincode

Aufbau von Hyperledger

Der **Chaincode** selber ist lediglich ein **Computerprogramm** welches ausgeführt werden kann **über eine REST API**. Beim Chaincode selber wird immer zuerst die *Invoke* Funktion aufgerufen. Diese Funktion entscheidet anhand der übergebenen *operationID* aus der REST API und der dazugehörigen Parameter, welche Funktion als nächstes aufgerufen werden soll. (später noch genauer bei Schnittstelle).

Die **Aufgabe der Blockchain** ist es lediglich zu speichern, wann welche Funktion mit welchen Parametern übergeben worden ist. Jeder Funktionsaufruf wird in einem neuen Block gespeichert. Da dass Programm deterministisch abläuft, kann bereits aus dieser Information der Zustand des kompletten aktuellen Programms berechnet werden.

Aus der Blockchain können nun alle relevanten Daten ausgelesen werden. Die SAP bietet hierfür bereits einen passenden *Block-Explorer* zu r Verfügung.

Man sieht genau, wann, welche, Funktion (in dem Fall die *returnACarById*) durch wen mit welchen Argumenten (Parametern) aufgerufen worden ist. Beziehungsweise, welche Funktion durch die *Invoke* Funktion aufgerufen worden ist. Ebenfalls kann man im Chaincode Events erstellen. Das sind mehr oder weniger Block Beschreibungen welche man optional erstellen kann um einen besseren Überblick zu bekommen

Als letztes beinhaltet Hyperledger noch einen sogenannten **World State**. Eine Key-Value Map, in der Daten/Objekte im Chaincode gespeichert und ausgelesen werden können. In genau dieser Map werden all unsere User, Autos, Fahrtenbücher... gespeichert - Beispiele folgen später. Ebenfalls werden alle Zugriffe jeglicher Art auf diesen World State in einem Block gespeichert.

Diese Information befindet sich ebenfalls in Block 242. Auf der rechten Seite stehen alle Lesezugriffe, auf der linken Seite alle Schreibzugriffe. Auf der rechten Seite sieht man, wie der *user2* und das *car2* aus dem World State heraus gelesen werden. Nun kann man diese beiden Objekte bearbeiten und anschließend den neuen Stand im World State abspeichern. In dem Fall wurde die *borrowId* bei beiden auf 0 gesetzt, was bedeutet dass nun beide wieder frei zum ausleihen sind. Ebenfalls wurde das *travelLog2* erstellt. Dies stellt das Fahrtenbuch dar. Es wurde komplett automatisch erstellt mit allen relevanten Daten und kann nun mit der Funktion *getTravelLog* ausgelesen und abgefragt werden.

Copyright © 2017, Fink IT-Solutions GmbH & Co.KG.

Schnittstelle

Wie bereits erwähnt handelt es sich bei der Schnittstelle um eine REST API welche selber definiert werden muss. Nützlich dazu ist das Tool Swagger[2]. In diesem wird der Code welchen man schreibt, direkt visuell angezeigt was beim Entwickeln eine große Hilfe ist.

Es wird hier nun nicht genauer darauf eingegangen wie man eine REST API programmiert. Es ist lediglich wichtig zu wissen dass alle *4 HTTP Requests* auf eine Ressource/Objekt verfügbar sein müssen, damit es sich um eine "richtige" REST API handelt. Es muss also gewährleistet sein das man eine Ressource wie z.B den User oder das Auto abfragen (**GET**), erstellen (**POST**), updaten (**PUT**) und löschen kann (**DELETE**). Zudem werden hier alle weiteren Möglichkeiten angeboten, welche der Chaincode behandeln kann, wie z.B das ein User ein Auto ausleihen und abgeben kann.

Jeder Pfad in der REST API hat zudem eine eigene ID, die sogennante *operationID.* Diese wird ebenfalls mitgegeben und kann im Chaincode mit der Methode *stub.GetFunctionAndParamter* abgefragt werden. Je nach *operationID* soll die dazugehörige Funktion im Chaincode aufgerufen werden, wie unten im Code zu sehen. In args werden die ganzen restlichen Parameter gespeichert, welche im HTTP header und body mitgeschickt werden.

Copyright © 2017, Fink IT-Solutions GmbH & Co.KG.

Copyright © 2017, Fink IT-Solutions GmbH & Co.KG.

```go
func (cc *CRUD) Invoke(stub shim.ChaincodeStubInterface) peer.Response {

    function, args := stub.GetFunctionAndParameters()

    switch strings.ToLower(function) {
    //CAR OPERATIONS
    case "createcar":
        return cc.createCar(stub, args)
    case "getcarbyid":
        return cc.getCar(stub, args)
    case "updatecar":
        return cc.updateCar(stub, args)
    case "deletecar":
        return cc.deleteCar(stub, args)
    //USER OPERATIONS
    case "createuser":
        return cc.createUser(stub, args)
    case "getuserbyid":
        return cc.getUser(stub, args)
    case "updateuser":
        return cc.updateUser(stub, args)
    case "deleteuser":
        return cc.deleteUser(stub, args)
    //BUSINESS OPERATIONS
    case "borrowacar":
        return cc.borrowACar(stub, args)
    case "getborrowlogbyid":
        return cc.getBorrowLogById(stub, args)
    case "returnacarbyid":
        return cc.returnACarById(stub, args)
    case "gettravellogbyid":
        return cc.getTravelLogById(stub, args)
    default:
        logger.Warningf("Invoke('%s') invalid!", function)
        return Error(http.StatusNotImplemented, "Invalid method name!!!")
    }
}
```

Somit können alle Funktionen, welche im Chaincode enthalten sind über eine dazugehörige REST API aufgerufen werden.

Datenmodellierung

Im Chaincode können *structs* definiert werden, welche vorschreiben wie ein Auto oder User aufgebaut sind. Autos und User haben jeweils eine eindeutige ID und zusätzliche Attribute wie Name oder Km (Kilometerstand). Bei dem Attribut BorrowId wird die ID des dazugehörigen CarBorrow gespeichert. Eine BorrowId=0 bedeutet dass das Auto gerade nicht ausgeliehen ist bzw der User gerade kein Auto ausgeliehen hat. Leiht der User ein Auto aus, wird das dazugehörige "CarBorrow" erstellt. In diesem stehen alle wichtigen Informationen über die Ausleihe und werden über die *BorrowId* des Autos/Users referenziert.

Copyright © 2017, Fink IT-Solutions GmbH & Co.KG.

```
type Car struct {
    Id       int `json:"id"`
    Km       int `json:"km"`
    BorrowId int `json:"borrowId"`
}
```

```
type User struct {
    Id       int    `json:"id"`
    Name     string `json:"name"`
    BorrowId int    `json:"borrowId"`
}
```

```
type CarBorrow struct {
    Id        int    `json:"id"`
    CarId     int    `json:"carId"`
    UserId    int    `json:"userId"`
    StartTime string `json:"startTime"`
}
```

```
type TravelLog struct {
    Id        int    `json:"id"`
    UserId    int    `json:"userId"`
    CarId     int    `json:"carId"`
    Usage     string `json:"usage"`
    StartKm   int    `json:"startKm"`
    EndKm     int    `json:"endKm"`
    DrivenKm  int    `json:"drivenKm"`
    StartTime string `json:"startTime"`
    EndTime   string `json:"endTime"`
}
```

In der REST API kann man dem Benutzer vorgeben, wie die Daten modelliert sein sollen(siehe nächstes Bild), damit diese vom Chaincode auch bearbeitet werden können. Letztendlich kann man über die REST API alles beliebige verschicken und der Chaincode validiert diese Daten dann auf ihre "Korrektheit". Daten sind in dem Sinne nur korrekt wenn man aus dem übergebenem JSON Objekt ein dazugehöriges *st ruct* konstruieren kann und alle Parameter einen Wert haben. Ist dies nicht der Fall, bekommt der Benutzer eine passende Fehlermeldung zurück.

Für das *CarBorrow* und *TravelLog*, gibt es nur eine dazugehörige *GET* Schnittstelle. Dass ist nicht REST konform, aber dafür Blockchain konform. Es ist die Aufgabe des Chaincodes, diese beiden Object alleine, vollautomatisiert, ohne menschliche Eingriffe zu erstellen, deshalb soll es auch nicht möglich sein diese Objekte selber zu erstellen oder zu bearbeiten.

Speichern und Auslesen aus dem World State

Unser Programm besteht aus 4 verschiedenen Objekten welche im World State gespeichert und ausgelesen können werden. Das Auto , der User , das Ausleih Object (CarBorrow) und das Fahrtenbuch (TravelLog). Alle 4 Objekte können über eine dazugehörige *GET* Funktion abgefragt werden. Jedoch erwarten alle 4 Funktionen lediglich eine ID. Wie ist es dann möglich dass wenn man die Funktion *getCar(1)* aufruf t, das Auto mit der ID=1 zurück bekommt und wenn man die Funktion *getTravelLog(1)* aufruft, das dazugehörige Fahrtenbuch mit der ID=1 zurück bekommt.

Objekt	prefix	übergebeneID	key im World State	
Car	"car"	"2"	"car2"	
User	"user"	"2"	"user2"	
CarBorrow	"borrow"	"2"	"borrow2"	
TravelLog	"travelLog"	"2"	"travelLog2"	

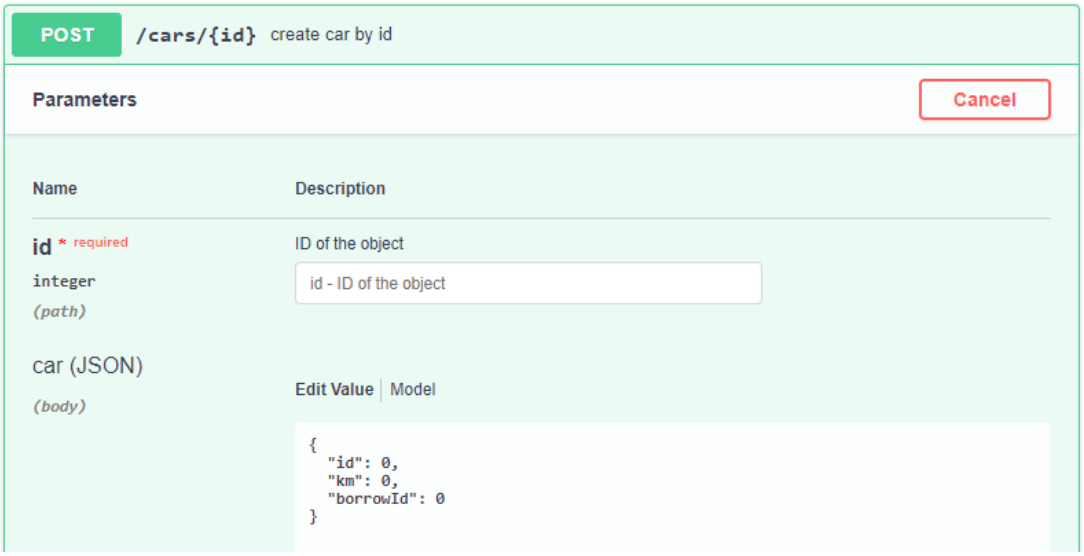
Copyright © 2017, Fink IT-Solutions GmbH & Co.KG.

Für jedes der 4 Objekte wird ein anderer eindeutiger **prefix** genommen als Schlüssel für den World State. Ein Beispiel folgt im nächsten Unterpunkt - Programmablauf

Programmablauf

Im folgenden versuchen wir ein neues Auto anzulegen und gehen somit in der REST API auf den dazugehörigen Pfad.

Im HTTP path (header?) wird immer die ID des jeweiligen Objektes verschickt, und im HTTP body, wird das Objekt als JSON verschickt.



Drückt man nun auf *execute* (nicht zu sehen auf dem Bild) werden die Daten an die createCar Funktion im Chaincode verschickt. Diese Funktion gehen wir nun einmal durch

Copyright © 2017, Fink IT-Solutions GmbH & Co.KG.

```
186 func (cc *CRUD) createCar(stub shim.ChaincodeStubInterface, args []string) peer.Response {
187     //(path) -> args[0]: "id"
188     //(body) -> args[1]: {"id":"string","km":"string","owner":"string"} JSON obj.
189
190     if obj, err := stub.GetState("car" + args[0]); err != nil || obj != nil {
191         return Error(http.StatusConflict, "a car with this id already exists")
192     }
193
194     //create a car with the overgiven parameters
195     var car Car
196     json.Unmarshal([]byte(args[1]), &car)
197
198     //check if the car has all three values
199     if car.Id == 0 || car.Km == 0 || car.BorrowId != 0 {
200         return Error(http.StatusBadRequest, "one parameter is wrong!")
201     }
202
203     //check if car.id is the same id as in path
204     erg, err := strconv.Atoi(args[0])
205     if car.Id != erg {
206         return Error(http.StatusBadRequest, "id of path and id of car are different!")
207     }
208
209     if err := stub.PutState("car"+args[0], []byte(args[1])); err == nil {
210         stub.SetEvent("Car created", []byte("Success"))
211         return Success(http.StatusCreated, "Ok", nil)
212     } else {
213         return Error(http.StatusInternalServerError, err.Error())
214     }
215 }
```

Fangen wir mit den übergebenen Parametern an. Der erste ist der *stub*. Das Paket shim[3] bietet alle Funktionen welche benötigt werden um Transaktionen zu analysieren, World State Daten auszulesen ....

in args befinden sich die eigentlichen Parameter - in Zeile 187 und 188 sieht man was sich darin alles befindet.

Zeile 190: Hole das Auto mit der übergebenen ID. Ist das Objekt was man zurück bekommt bereits vorhanden - existiert ein Auto mit dieser ID bereits und eine entsprechende Fehlermeldung wird zurückgegeben

Zeile 195 + 196: erstelle ein Auto und initialisiere es mit den übergebenen Daten

Zeile 199: Überprüfe ob alle Werte "richtig" sind

Zeile 209: Wenn alles OK ist, schreibe das neue übergebene Auto in den World State mit dem Schlüssel "car(übergebene ID)"

Zeile 210: Erstelle ein passendes Event

Zeile 211 / 213: return ein Erfolg oder Error

WICHTIG:

Nur wenn ein Success zurück gegeben wird, wird diese Transaktion auch in die Blockchain übernommen. Wird ein Error zurückgegeben, ist das alles was passiert. In dem Fall werden auch keine Änderungen im World State übernommen oder Events erstellt

Quellen

1 - https://help.sap.com/viewer/cab6d96064454f348689e661d3fe569b/BLOCKCHAIN/en-US/cc28ff191f6142e7965614f437a34dc2.html

Copyright © 2017, Fink IT-Solutions GmbH & Co.KG.

2 - https://editor.swagger.io/

3 - https://godoc.org/github.com/hyperledger/fabric/core/chaincode/shim

Copyright © 2017, Fink IT-Solutions GmbH & Co.KG.