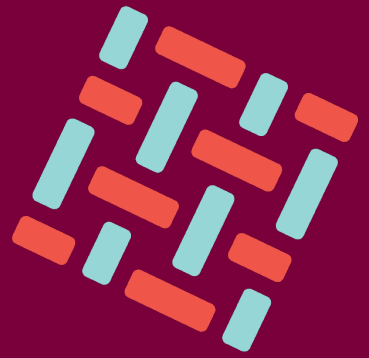
Vilnius
University

Installing and using Hyperledger Fabric



HYPERLEDGER
FABRIC

Prof. dr. Remigijus Paulavičius & dr. Ernestas Filatovas, 2022

Outline

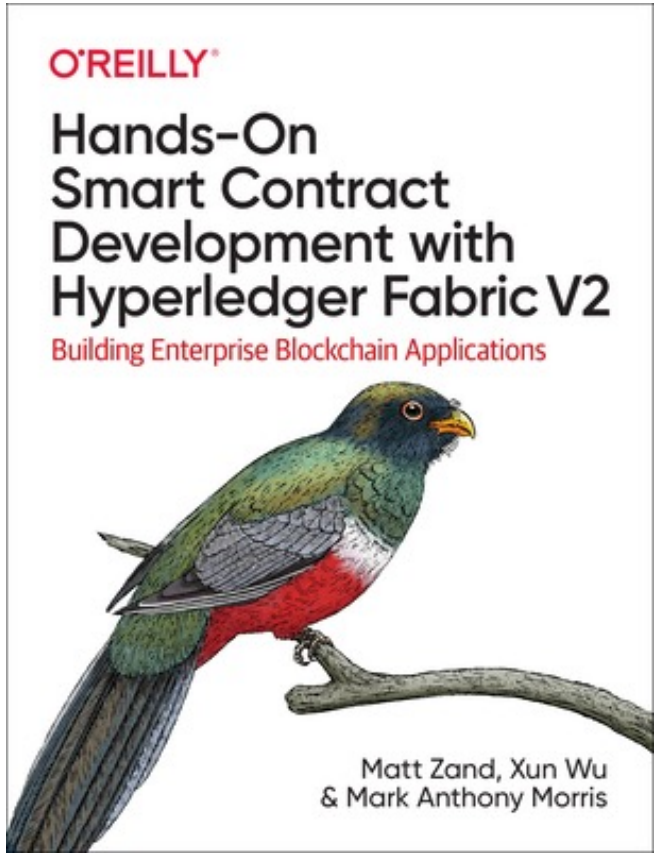
1. Prerequisites
2. Installing Hyperledger Fabric binaries and sample projects
3. Fabcar smart contract (Fabcar.js)
4. Fundamental Requirements of a Smart Contract
5. Fabcar client, Fabric test network
6. Invoking Smart Contract Transactions
7. Fabric SDK for Node.js Command-Line Application (Fabcar smart contract client)

Goals of this lecture

You will learn about Fabric smart contract development by examining a simple smart contract and the Fabric APIs used to implement Fabric smart contracts.

This lecture will help you achieve the following practical goals:

- Writing a Fabric smart contract by using the JavaScript programming language
- Installing and instantiating a Fabric smart contract
- Validating and sanitizing inputs and arguments in a smart contract
- Creating and running simple or complex queries



Lecture material

- We are going to use the Hyperledger-provided **binaries** and **sample projects** from the **Fabric project**.
- These binaries and sample projects will help us **start a Fabric test network**, and the **sample projects provide several example smart contracts** from which to **learn how to develop your own**.
- In this lecture **we will examine an example smart contract** from a sample project called **Fabcar**.
- The **binaries** we use **have the same name** on all supported operating systems.

Prerequisites

Vilnius University

(Installing) Prerequisites

- Before we can develop Fabric smart contracts, we need to **download and install the software** required **to download Hyperledger Fabric**.
- To download and set up Hyperledger Fabric for developing Fabric smart contracts, **we will execute a script that requires certain software to exist on the platform** you are developing on — **Windows, Linux, or Mac**.
- **We need to install**:
  - Git,
  - cURL,
  - Node.js,
  - npm,
  - Docker and Docker Compose,
  - Fabric installation script

Install Git, cURL

- Git is used to **clone the fabric-samples** repository **from GitHub** to your local machine.
- If you don't have Git installed, you can **download it from** https://git-scm.com/downloads
- Once you download and install it, verify Git installation with the following command:


```
$ git --version
git version 2.33.0
```

- We use **cURL to download the Fabric binaries from the web**. You can download cURL from https://curl.haxx.se/download.html
- Once it's downloaded and installed, verify the installation:


```
$ curl -V
curl 7.77.0 (x86_64-apple-darwin21.0) libcurl/7.77.0 (SecureTransport) LibreSSL/2.8.3 zlib/1.2.11
nghttp2/1.42.0
```

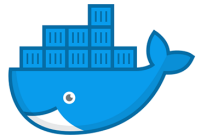
Install Node.js and npm

- We will be using **JavaScript** for developing our Fabric smart contracts.
- **Fabric uses Node.js and npm for processing smart contracts**.
- The **supported versions** of **Node.js are 10.15.3 and higher**, and **12.13.1 and higher**.
- The supported versions of npm are **6 and higher**.
- **Node.js includes npm in the installation.**
- You can download Node.js from https://nodejs.org/en/download
- You can verify the installation of Node.js and npm by executing the following commands:


```
$ node -v
v17.0.1

$ npm -v
8.1.4
```

Install Docker and Docker Compose



- **Hyperledger Fabric consists of several components, each of which operates as a separate executable service, so Fabric maintains Docker images of each component.**
- The images are hosted on the official Docker Hub website. **At minimum, you need Docker version 17.06.2-ce.**
- You can get the latest version of Docker at https:// www.docker.com/get-started
- **When Docker is installed, Docker Compose is also installed.**
- You can verify the Docker and Docker Compose versions by executing following commands:

```
$ docker -v
Docker version 20.10.10, build b485636

$ docker-compose --version
Docker Compose version v2.1.1
```

Installing Hyperledger Fabric binaries and sample projects

Vilnius University

Fabric Installation Script

HYPERLEDGER FABRIC

- **Before proceeding, start Docker, because Docker needs to be running to complete the installation of the Fabric installation script!!!**
- **Create and change to the directory** you will use **to install the Fabric binaries and sample projects**.
- The script will do the following:
  1. **Download the Fabric binaries**
  2. **Clone fabric-samples from the GitHub repo**
  3. **Download the Hyperledger Fabric Docker images**

```
$ docker -v
Docker version 20.10.10, build b485636

$ docker-compose --version
Docker Compose version v2.1.1
```

Fabric Installation Script

- **Before proceeding, start Docker, because Docker needs to be running to complete the installation of the Fabric installation script!!!**
- **Create and change to the directory (e.g. Fabric** or **HyperFabric)** you will use **to install the Fabric binaries and sample projects**.
- Execute the following command:

```
$ curl -sSL https://bit.ly/2ysbOFE > FabricDevInstall.sh
```

- Now you can open **FabricDevInstall.sh** in your favorite editor and examine the script to see how it **clones fabric-samples from GitHub**, **downloads the Fabric binaries**, and **downloads the Docker images**.
- Understanding the operation of this script may help you later if you want to customize your Fabric development environment or optimize it based on your workflow.

FabricDevInstall.sh (1)

#!/bin/bash

 # dockerPull() pulls docker images from fabric and chaincode repositories

 # note, if a docker image doesn't exist for a requested release, it will simply

 # be skipped, since this script doesn't terminate upon errors.

```
32  dockerPull() {
33      #three_digit_image_tag is passed in, e.g. "1.4.7"
34      three_digit_image_tag=$1
35      shift
36      #two_digit_image_tag is derived, e.g. "1.4", especially useful as a local tag for two digit references to most recent baseos, ccenv
37      two_digit_image_tag=$(echo "$three_digit_image_tag" | cut -d'.' -f1,2)
38      while [[ $# -gt 0 ]]
39      do
40          image_name="$1"
41          echo "====> hyperledger/fabric-$image_name:$three_digit_image_tag"
42          docker pull "hyperledger/fabric-$image_name:$three_digit_image_tag"
43          docker tag "hyperledger/fabric-$image_name:$three_digit_image_tag" "hyperledger/fabric-$image_name"
44          docker tag "hyperledger/fabric-$image_name:$three_digit_image_tag" "hyperledger/fabric-$image_name:$two_digit_image_tag"
45          shift
46      done
47  }
```

FabricDevInstall.sh (2)

- *# clone (if needed) **hyperledger/fabric-samples** and checkout*
- *# corresponding version to the binaries and docker images to be downloaded*

```
cloneSamplesRepo() {
    # clone (if needed) hyperledger/fabric-samples and checkout corresponding
    # version to the binaries and docker images to be downloaded
    if [ -d test-network ]; then
        # if we are in the fabric-samples repo, checkout corresponding version
        echo "==> Already in fabric-samples repo"
    elif [ -d fabric-samples ]; then
        # if fabric-samples repo already cloned and in current directory,
        # cd fabric-samples
        echo "===> Changing directory to fabric-samples"
        cd fabric-samples
    else
        echo "===> Cloning hyperledger/fabric-samples repo"
        git clone -b main https://github.com/hyperledger/fabric-samples.git && cd fabric-samples
    fi

    if GIT_DIR=.git git rev-parse v${VERSION} >/dev/null 2>&1; then
        echo "===> Checking out v${VERSION} of hyperledger/fabric-samples"
        git checkout -q v${VERSION}
    else
        echo "fabric-samples v${VERSION} does not exist, defaulting to main. fabric-samples main branch is intended to work with recent
        git checkout -q main
    fi
}
```

Execute FabricDevInstall.sh

- After you finish examining the script, **open a shell and change FabricDevInstall.sh to an executable** by executing the following command:

```
$ chmod +x FabricDevInstall.sh
```

- Now let's execute FabricDevInstall.sh with the following command:

```
$ ./FabricDevInstall.sh
```

- **Once the script completes execution, we should be all set.**
- The **fabric-samples** directory, **Docker images**, and **Fabric binaries** are installed in the **fabric-samples/bin** directory.
- In the directory where you ran the command, there should now be a directory called **fabric-samples**. Everything we need is in fabric-samples.
- First, we will dig into the **Fabcar sample smart contract**, which you can find in the **fabric-samples directory**.

Fabcar smart contract (Fabcar.js)

Vilnius University

Fabcar smart contract (1)

SMART CONTRACT

- Let's start with the simple Fabcar smart contract.
- In the **fabric-samples** directory, locate the **chaincode** directory.
- In the **chaincode** directory, locate the **fabcar** directory.
- In the **fabcar** directory, locate the **javascript** directory.

```
$ cd chaincode/fabcar/javascript
```

- Change to this directory in your shell and execute the following command:

```
$ npm install
```

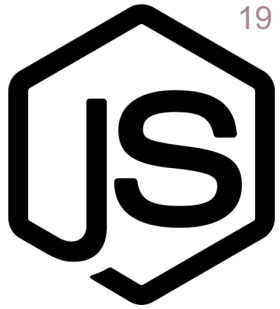
- This will create the **node_modules** directory and install the dependent modules defined in **package.json**.
- We did this because we depend on the **fabric-contract-api** and **fabric-shim** modules. These are the two modules we use when developing Fabric smart contracts in JavaScript.
- We will look at these after we examine the Fabcar smart contract.

Fabcar smart contract (2)

SMART CONTRACT

- Now let's examine the Fabcar smart contract.
- This simple smart contract is a great example for learning Fabric smart contract development because it contains necessary details to form a foundation from which we can move on to more advanced smart contracts.
- **It is located in the lib directory**, which should be: **fabric-samples/chaincode/fabcar/javascript/lib/** directory.
- Open **fabcar.js** in your favorite editor.

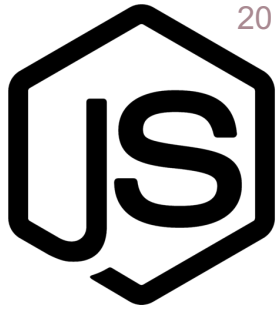
Fabcar.js (1)



1. We start by importing the **fabric-contract-api** module (see line 9).

```
1  /*
2   * Copyright IBM Corp. All Rights Reserved.
3   *
4   * SPDX-License-Identifier: Apache-2.0
5   */
6
7  'use strict';
8
9  const { Contract } = require('fabric-contract-api');
```

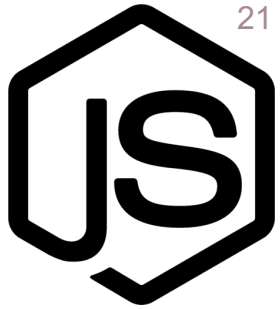
Fabcar.js (2)



2. All Fabric smart contracts **extend the Contract class** (see line 11).
   - We get the Contract class from the **fabric-contract-api** module we imported in line 9.
3. **Smart contracts can use transactions to initialize them** prior to processing client application requests.
   - Line 13 is the **beginning of the function** (**initLedger**) that initializes the smart contract.
   - **All smart contract functions receive a transaction context object** as an argument, called by convention **ctx**.

```
11  class FabCar extends Contract {
12
13      async initLedger(ctx) {
14          console.info('============ START : Initialize Ledger ==========');
```

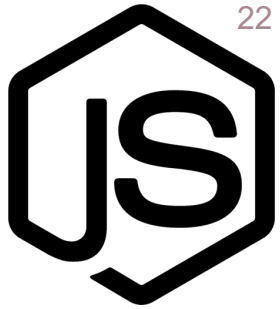
Fabcar.js (3)



4. In this example, the initLedger function is creating an **array of objects called cars**.
   - **Each array object contains key-value pairs**.
   - You can think of the array of objects as **records of assets**, and the object **key-value pairs as the fields**.
   - This function effectively preloads an array of car objects for exercising the transaction functions in the smart contract.

```
15          const cars = [
16              {
17                  color: 'blue',
18                  make: 'Toyota',
19                  model: 'Prius',
20                  owner: 'Tomoko',
21              },
22              {
23                  color: 'red',
24                  make: 'Ford',
25                  model: 'Mustang',
26                  owner: 'Brad',
27              },
28              {
29                  color: 'green',
30                  make: 'Hyundai',
31                  model: 'Tucson',
32                  owner: 'Jin Soo',
33              },
34              {
```

Fabcar.js (4)

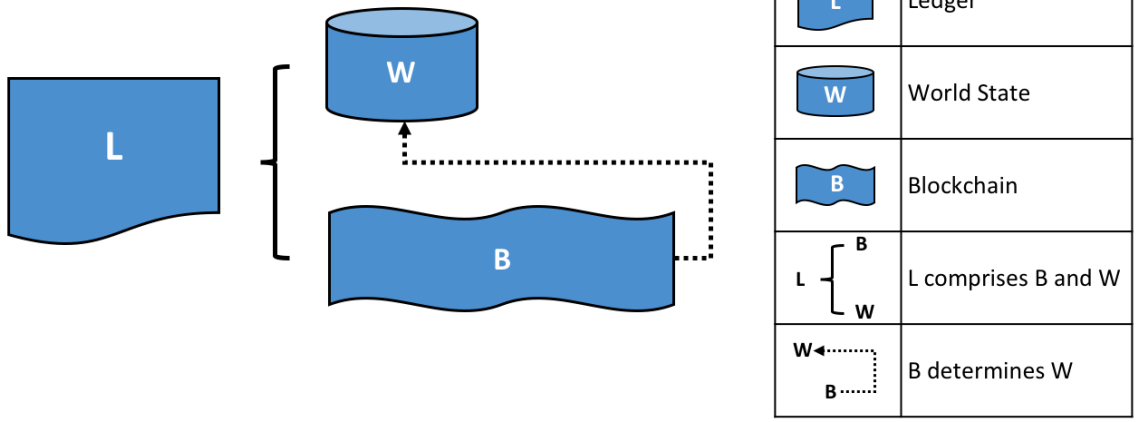


5. Next, the **initLedger** function is iterating (**using for loop**) through the array of car asset objects (see Line 78) and adding a field called **docType** to each object (see Line 79), and assigning the string value **car** to each object.
6. Line 80 is the first use of the **ctx** object (**Context** class) **passed as the first function argument to all Contract class transaction functions**. The **ctx object contains the stub object**, which is a **ChaincodeStub** class.
   - The **ChaincodeStub** class implements an API to access the ledger. This line calls the **putState** function, which **writes the key and value to the ledger and world state**.

```
78          for (let i = 0; i < cars.length; i++) {
79              cars[i].docType = 'car';
80              await ctx.stub.putState('CAR' + i, Buffer.from(JSON.stringify(cars[i])));
81              console.info('Added <--> ', cars[i]);
82          }
83          console.info('============ END : Initialize Ledger ==========');
```

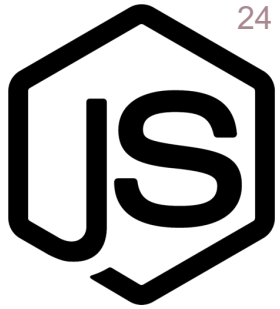
Ledger

- **Hyperledger Fabric implements the blockchain ledger in two components**: a **file-based component** and a **database component**.
- The file-based component (**Blockchain**) is the low-level immutable data structure implementing the ledger, and the database component exposes the current state of the file-based ledger.
- The **database component is called the world state** because it represents the current state of the ledger.
- The file-based component maintains the perpetual immutable ledger.
- The **fabric-contact-api** accesses the world state.
- Lower-level APIs access the filebased ledger.



https://hyperledger-fabric.readthedocs.io/en/release-2.2/ledger/ledger.html

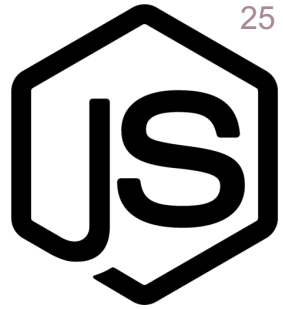
Fabcar.js (5)



7. The **first transaction function** (**queryCars**) comes in Line 86. As stated earlier, the first argument of all smart contract transaction functions is the **ctx** object, which represents the transaction context and is a Context class. Any other arguments are optional.

```
86      async queryCar(ctx, carNumber) {
87          const carAsBytes = await ctx.stub.getState(carNumber); // get the car from chaincode state
88          if (!carAsBytes || carAsBytes.length === 0) {
89              throw new Error(`${carNumber} does not exist`);
90          }
91          console.log(carAsBytes.toString());
92          return carAsBytes.toString();
93      }
```

Fabcar.js (6)



8. The **queryCars** function **is a read transaction**.
   - Using the **ctx** object, it **calls** the **stub's getState** function (Line 87), which will **read from the world state**—the database. The **stub** is an implementation of the **ChaincodeStub** class, which we will cover later.
   - For this function, the argument called **carNumber** is the key passed to the **getState** function, which **will search the world state database for the key and return the associated value stored for i**t.
   - The remainder of the function **checks whether data was returned** (Line 88) and, if so, **converts the byte array returned into a string and returns the string that represents the value of the key stored in the world state** (Line 89).
   - **Remember**, the world state is a representation of the perpetual immutable file-based ledger's current state for any key-value pair stored in the ledger. While the database may be mutable, the file-based ledger is not. **So even when a key-value pair is deleted from the database or world state, the key-value pair is still in the file-based ledger where all history is maintained**.

```
86      async queryCar(ctx, carNumber) {
87          const carAsBytes = await ctx.stub.getState(carNumber); // get the car from chaincode state
88          if (!carAsBytes || carAsBytes.length === 0) {
89              throw new Error(`${carNumber} does not exist`);
90          }
91          console.log(carAsBytes.toString());
92          return carAsBytes.toString();
93      }
```

Fabcar.js (7)

9. Then we have the second transaction function (**createCar**). It **passes the values required to create a new car record object**, which we will add to the ledger.

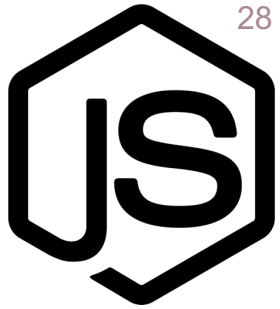
```
95      async createCar(ctx, carNumber, make, model, color, owner) {
96          console.info('============ START : Create Car ===========');
97
98          const car = {
99              color,
100             docType: 'car',
101             make,
102             model,
103             owner,
104         };
105
106         await ctx.stub.putState(carNumber, Buffer.from(JSON.stringify(car)));
107         console.info('============ END : Create Car ===========');
108     }
```

Fabcar.js (8)

10. With the **car** record object built from the function arguments, we call the Chain codeStub API function implemented by **stub**, called **putState**, which **will write the key and value to the ledger and update the current world state**.
    - The first two arguments that pass to the **putState** function are a **key** (carNumber) and a **value (car object)**, respectively. We need to change the value, the car record object, into a byte array, which ChaincodeStub APIs require.

```
95      async createCar(ctx, carNumber, make, model, color, owner) {
96          console.info('============= START : Create Car ===========');
97
98          const car = {
99              color,
100             docType: 'car',
101             make,
102             model,
103             owner,
104         };
105
106         await ctx.stub.putState(carNumber, Buffer.from(JSON.stringify(car)));
107         console.info('============= END : Create Car ===========');
108     }
```

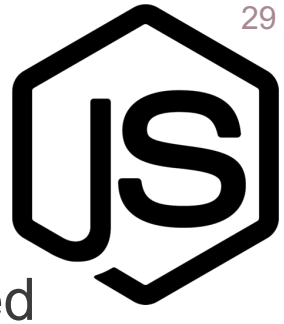
Fabcar.js (9)



11. The next transaction function, called **queryAllCars**, is a read transaction and demonstrates a **range query**.
    - A range query, like all queries, is executed by the peer that receives the request.
    - A range query takes two arguments: the **beginning key** and the **ending key**.
    - These two keys represent the beginning and end of the range. **All keys that fall into the range are returned along with their associated values**.
    - You **can pass an empty string for both keys to retrieve all keys and values**.

```
110     async queryAllCars(ctx) {
111         const startKey = '';
112         const endKey = '';
113         const allResults = [];
114         for await (const {key, value} of ctx.stub.getStateByRange(startKey, endKey)) {
115             const strValue = Buffer.from(value).toString('utf8');
116             let record;
117             try {
118                 record = JSON.parse(strValue);
119             } catch (err) {
120                 console.log(err);
121                 record = strValue;
122             }
123             allResults.push({ Key: key, Record: record });
124         }
125         console.info(allResults);
126         return JSON.stringify(allResults);
127     }
```

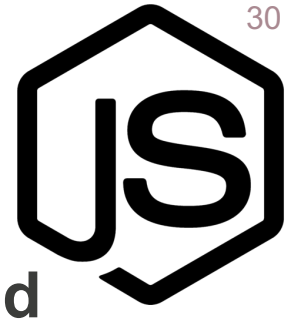
Fabcar.js (10)



12. A for **loop** is executed (see Line 114), which stores all the keys and associated values returned from the **ChaincodeStub API** function **getStateByRange**

```
110     async queryAllCars(ctx) {
111         const startKey = '';
112         const endKey = '';
113         const allResults = [];
114         for await (const {key, value} of ctx.stub.getStateByRange(startKey, endKey)) {
115             const strValue = Buffer.from(value).toString('utf8');
116             let record;
117             try {
118                 record = JSON.parse(strValue);
119             } catch (err) {
120                 console.log(err);
121                 record = strValue;
122             }
123             allResults.push({ Key: key, Record: record });
124         }
125         console.info(allResults);
126         return JSON.stringify(allResults);
127     }
```

Fabcar.js (11)



13. The last transaction function, **changeCarOwner**, **combines both read and write tasks** to change the world state.
    - **The business logic here is a transfer of ownership**. In addition to the **ctx** argument, two arguments are passed: a **key** called **carNumber**, and a **value** object called **newOwner**.

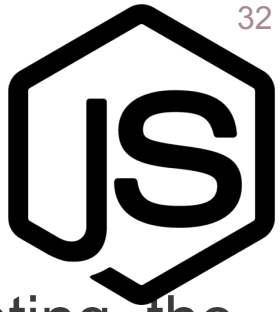
```
129     async changeCarOwner(ctx, carNumber, newOwner) {
130         console.info('============ START : changeCarOwner ===========');
131
132         const carAsBytes = await ctx.stub.getState(carNumber); // get the car from chaincode state
133         if (!carAsBytes || carAsBytes.length === 0) {
134             throw new Error(`${carNumber} does not exist`);
135         }
136         const car = JSON.parse(carAsBytes.toString());
137         car.owner = newOwner;
138
139         await ctx.stub.putState(carNumber, Buffer.from(JSON.stringify(car)));
140         console.info('============ END : changeCarOwner ===========');
141     }
```

Fabcar.js (12)

14. Next, we need to retrieve the record object from the world state, which represents the current key and value for this record.
    - The key is **carNumber** (see Line 132). We use it to execute the **ChaincodeStub API getState**.
    - Once we retrieve the current car record object for **carNumber**, we change the owner field to **newOwner**.

```
129     async changeCarOwner(ctx, carNumber, newOwner) {
130         console.info('============= START : changeCarOwner ===========');
131
132         const carAsBytes = await ctx.stub.getState(carNumber); // get the car from chaincode state
133         if (!carAsBytes || carAsBytes.length === 0) {
134             throw new Error(`${carNumber} does not exist`);
135         }
136         const car = JSON.parse(carAsBytes.toString());
137         car.owner = newOwner;
138
139         await ctx.stub.putState(carNumber, Buffer.from(JSON.stringify(car)));
140         console.info('============= END : changeCarOwner ===========');
141     }
```

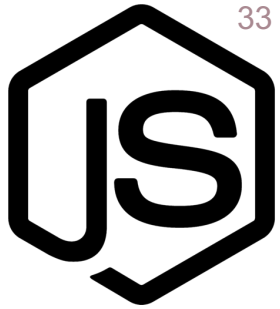
Fabcar.js (13)



15. After retrieving the ledger data representing the world state and updating the retrieved data, we update the ledger for this car record object by executing the **ChaincodeStub API putState** (see Line 139).
    - This writes a new key and value to the ledger that represents the world state.

```
129     async changeCarOwner(ctx, carNumber, newOwner) {
130         console.info('============ START : changeCarOwner ===========');
131
132         const carAsBytes = await ctx.stub.getState(carNumber); // get the car from chaincode state
133         if (!carAsBytes || carAsBytes.length === 0) {
134             throw new Error(`${carNumber} does not exist`);
135         }
136         const car = JSON.parse(carAsBytes.toString());
137         car.owner = newOwner;
138
139         await ctx.stub.putState(carNumber, Buffer.from(JSON.stringify(car)));
140         console.info('============ END : changeCarOwner ===========');
141     }
```

Fabcar.js (14)



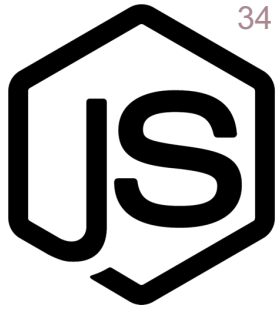
15. After retrieving the ledger data representing the world state…
    - If the **car record object is now retrieved, the ledger will not show the new owner until the record object is committed to the ledger**.
    - **It is important to understand that once committed, the ledger is appended, and the database state will be changed (the world state will be updated)**.
    - You can think of the ledger as an ever-growing stack of objects, each with a unique identifier called the key.
    - There can be many keys with the same value, but only one represents the current or world state.
    - This is how the **database implements the view of the world state**, **while the file-based ledger implements the immutable history of all keys in timestamp order**.
    - **The file-based ledger stores all write transactions.**

      Both successful and unsuccessful write transactions are part of the filebased immutable ledger.

      Flags control the validity of transactions stored in the immutable file-based ledger.

      This facilitates an audit of all submitted write transactions.

Fabcar.js (15)



16. Line 145 is Node.js specific.
    - We discuss exporting smart contract modules in soon, when we cover smart contract execution, including project structure, packaging, and deployment.

```
145    module.exports = FabCar;
```

- This completes this simple smart contract.
- **We will now discuss it, in summary, to point out the fundamental requirements to develop a smart contract**.
- From this basic smart contract, complex smart contract applications can be designed and developed

Fundamental Requirements of a Smart Contract

Vilnius University

Fundamental Requirements of a Smart Contract (1)

- The **Fabcar example smart contract** is a valid, functional example of a **basic smart contract**.
- **We have much more to add before it would be ready for production**, including security, error management, reporting, monitoring, and testing.
- **We want to remember several important points from this example smart contract**.
- Let's go through them:

Fundamental Requirements of a Smart Contract (2)

- **Contract *class***
  - *Smart contracts **extend the Contract class**. This is a simple class with few functions.*
- ***Transaction context***
  - *All smart contract transaction functions **pass a transaction context object as their first argument**. This transaction **context object is a Context class**.*
- ***Constructor***
  - ***All smart contracts must have a constructor**. The **constructor argument is optional** and represents the name of the contract. If not passed, the class name will be used. We recommend you pass a unique name and think of this in terms of a namespace, like a reverse domain name structure.*
- ***Transaction function***
  - ***A transaction function to initialize a smart contract can be created and called prior to client requests**. You can use this to set up and execute your smart contract with any required resources. These resources could be tables or maps of data used for lookups, translations, decoding, validating, enriching, security, and so forth.*

Fundamental Requirements of a Smart Contract (3)

- ***World state***
  - *We can query the world state in multiple ways. The simple query is a key lookup, and a range query gets a set. There is another called a rich query. We look at world state in this lecture later.*
- **putState**
  - *To write data to the ledger, we use the **putState** function. It takes as arguments a key and value. **The value is a byte array, so the ledger can store any data**. Typically, we will store the equivalent of business objects that are marshaled into byte arrays prior to being passed as the value argument*.
- **ChaincodeStub**
  - The ChaincodeStub class contains several functions used to interact with the ledger and world state. All smart contracts get an implementation of this class as the **stub** object contained and exposed by the Context class implementation called **ctx**, which all transaction functions receive as their first argument
- ***Read/write transactions***
  - ***An update in a smart contract is executed in three steps: a read transaction, an update to the in-memory data returned from the read transaction, followed by a write transaction**. This creates a new world state for the key while maintaining the history of the key in the immutable file-based ledger.*
  - The important point to remember: you cannot update (or write to) the ledger and read back what you wrote **in the same transaction**.

Data flow of a transaction request

- You need to think about the **data flow of a transaction request**:
  1. **Clients submit transaction requests to peers, which endorse the request transaction (this is where the smart contract executes);**
  2. **the endorsements with read and write sets are sent back to clients;**
  3. **endorsed requests are sent to an orderer, which orders the transactions and creates blocks.**
  4. **The orderer sends the ordered requests to commit peers, which validate the read and write sets prior to committing the writes to the ledger and updating the world state.**

Summary of Requirements of a Smart Contract

- In the simplest form, a **smart contract is a wrapper around ChaincodeStub**, because smart contracts must use the interface exposed through this class to interact with the ledger and world state.
- This is an important point to remember.
- You should consider implementing business logic in a modular design, treating your Contract subclass like a datasource.
- This will facilitate evolving your code over time and partitioning logic into functional components that can be shared and reused.
- **In the second part of this lecture, we look into design in the context of packaging and deploying.**
- In the next lecture, we delve into modular design and implementation to facilitate maintenance and testing.

SDK (Software Development Kit) (1)

- Today the architecture places the smart contracts behind a gateway, which is middleware in the smart contract SDK.
- Fabric **provides an SDK implemented in Go, Java, and Node.js (JavaScript) for developing smart contracts**.
- We are interested in the Hyperledger Fabric smart contract development SDK for Node.js, which is called **fabric-chaincode-node**.
- While **you do not need to download it for smart contract development**, you can download or clone **fabric-chaincode-node** from GitHub (https://github.com/hyperledger/fabric-chaincode-node).

SDK (Software Development Kit) (2)

- The fabric-chaincode-node SDK has a lot going on.
- We are interested in a few components that are central to developing smart contracts.
- The remaining files are low-level interfaces, support artifacts, tools, and more required to implement the Contract interface with Node.js.
- This **SDK helps developers like us by providing a high-level API** so we can learn fast and focus on our smart contract business logic and design.
- The first API we are interested in is **fabric-contract-api** (already saw in Fabcar.js)
  - It is located under the apis subdirectory of **fabric-chaincode-node**.
- The other API you see, **fabric-shim-api**, is the type definition and pure interface for the fabric-shim library, which we look at later in this chapter.

SDK (Software Development Kit) (3)

- When we start our smart contract project and execute **npm install**, **npm** will download **fabric-contract-api as a module** from the npm public repository as well as **fabric-shim**.
- This download happens because we have two explicit smart contract dependencies for developing Hyperledger Fabric smart contracts.
- These are displayed in this excerpt from the **package.json** file of the Fabcar smart contract:

```
19      "dependencies": {
20          "fabric-contract-api": "^2.0.0",
21          "fabric-shim": "^2.0.0"
22      },
```

- **fabric-contract-api** and **fabric-shim** are the only modules we needed to develop our smart contracts. **fabric-contract-api** contains the **contract.js** and **context.js** files, which implement the Contract and Context classes.

```
$ find . -name contract.js
./node_modules/fabric-contract-api/lib/contract.js
```

Fabcar client, Fabric test network

Vilnius University

Practical goals of this part

1. Invoking smart contracts via the command-line interface
2. Evaluating transaction functions for queries
3. Submitting transactions and query transaction history
4. Creating and issuing an application contract

Fabric-samples (1)

- Installing the **Prerequisites** included **fabric-samples**, which is Hyperledger Fabric's project for helping developers learn Fabric smart contract development.
- It contains many smart contract samples and tools for the rapid development of Fabric smart contracts.
- **Included is a complete development Fabric network comprising two organizations.**
  - Each organization has a peer and CA.
  - The network is containerized using Docker, which makes it easy and fast to launch and manage

Fabric-samples (2)

- The **bin/ subdirectory** contains the **peer executable**, which we will use as **the first method to invoke the Fabcar smart contract**.
- The **chaincode/ subdirectory contains the Fabcar smart contract** (we already seen this), and the **fabcar/ directory** contains the **command-line client for the Fabcar smart contract**.
- We will use **test-network/** to **deploy the fabcar smart contract**.
- Once the contract is deployed, we can invoke it.

Fabcar client (1)

- Fabcar has two parts.
  - The **first part** is the **Fabcar smart contract**, which we examined already.
  - The **other part** is the **Fabcar client** (located in **fabcar/**). The client invokes the smart contract.
- The Fabcar client has **multiple implementations**. **We are interested in** the **JavaScript** implementation

```
~/HyperFabric/fabric-samples/fabcar on  main  22:47:20
$ ll
total 24
drwxr-xr-x   7 remigijus  staff   224B Dec  9 13:11 go
drwxr-xr-x   6 remigijus  staff   192B Dec  9 13:11 java
drwxr-xr-x  12 remigijus  staff   384B Dec  9 13:11 javascript
-rwxr-xr-x   1 remigijus  staff   324B Dec  9 13:11 networkDown.sh
-rwxr-xr-x   1 remigijus  staff   4.2K Dec  9 13:11 startFabric.sh
drwxr-xr-x   9 remigijus  staff   288B Dec  9 13:11 typescript
```

Fabcar client (2)

- Provided are **two shell scripts**, **startFabric.sh** and **networkDown.sh**, to **start** and **stop** our **Fabcar test network**.
- The **startFabric.sh** script prepares and runs a new environment; then it **deploys and initializes our Fabcar smart contract**.
- The **networkDown.sh** script shuts down all Docker containers, removes them, and cleans up the environment.
- **When the Fabric network restarts, it typically cleans all of the data**.
  - If you need to persist any work that's in a container that's part of your Fabcar test network, you need to persist your data prior to starting or stopping the test network.
  - Typically, you need to mount a volume for the directory **/var/hyperledger/production** in the orderer and peer Docker containers.

Fabcar client (3)

- The **Fabcar client is implemented as four Node.js command-line applications**, highlighted in Figure below.
  - Two of the applications, **enrollAdmin.js** and **registerUser.js**, **handle enrolling an administrator** and **registering an application user**.
  - The other two Node.js applications, **invoke.js** and **query.js**, handle an **invoke transaction** and a **query evaluation**, respectively.
- These are the four Node.js applications we will examine and execute to invoke the Fabcar smart contract

```
~/HyperFabric/fabric-samples/fabcar/javascript on  main  22:55:59
$ ll *.js
-rw-r--r--  1 remigijus  staff   2.1K Dec  9 13:11 enrollAdmin.js
-rw-r--r--  1 remigijus  staff   2.1K Dec  9 13:11 invoke.js
-rw-r--r--  1 remigijus  staff   2.1K Dec  9 13:11 query.js
-rw-r--r--  1 remigijus  staff   2.8K Dec  9 13:11 registerUser.js
```

Fabric test network

- The **Fabric test network** will launch a working Fabric network that **can be used for smart contract deployment**.
- The **network.sh** script is used indirectly when we use the Fabcar scripts **startFabric.sh** and **networkDown.sh** to **start** and **stop** the **test network**, respectively.
- This script is modular and can be refactored into a set of granular tools, suitable to your workflow and style.
- It can also serve as **a good example for learning how to develop** a script or scripts to launch and manage **your own custom Fabric network**.
- **For our development, it is a quick start and takes the pain out of designing, configuring, and managing all the resources required to implement a functional Fabric network**.

Understanding the startFabric.sh (1)

- The default implementation language is Go (see Line 13)

```
10  # don't rewrite paths for Windows Git Bash users
11  export MSYS_NO_PATHCONV=1
12  starttime=$(date +%s)
13  CC_SRC_LANGUAGE=${1:-"go"}
14  CC_SRC_LANGUAGE=`echo "$CC_SRC_LANGUAGE" | tr [:upper:] [:lower:]`
15
16  if [ "$CC_SRC_LANGUAGE" = "go" -o "$CC_SRC_LANGUAGE" = "golang" ] ; then
17    CC_SRC_PATH="../chaincode/fabcar/go/"
18  elif [ "$CC_SRC_LANGUAGE" = "javascript" ]; then
19    CC_SRC_PATH="../chaincode/fabcar/javascript/"
20  elif [ "$CC_SRC_LANGUAGE" = "java" ]; then
21    CC_SRC_PATH="../chaincode/fabcar/java"
22  elif [ "$CC_SRC_LANGUAGE" = "typescript" ]; then
23    CC_SRC_PATH="../chaincode/fabcar/typescript/"
24  else
25    echo The chaincode language ${CC_SRC_LANGUAGE} is not supported by this script
26    echo Supported chaincode languages are: go, java, javascript, and typescript
27    exit 1
28  fi
```

Understanding the startFabric.sh (2)

- You can choose another supported implementation language, like **JavaScript**, by supplying a supported argument as shown here:

```
$ ./startFabric.sh javascript
```

- After setting the implementation language, script runs the following commands:

```
30    # clean out any old identites in the wallets
31    rm -rf javascript/wallet/*
32    rm -rf java/wallet/*
33    rm -rf typescript/wallet/*
34    rm -rf go/wallet/*
```

- As you can see, **all implementations have their wallet subdirectory contents removed**. We will discuss wallets when we execute the Fabcar client applications.

Understanding the startFabric.sh (3)

- Finally, the script executes:

```
36  # launch network; create channel and join peer to channel
37  pushd ../test-network
38  ./network.sh down
39  ./network.sh up createChannel -ca -s couchdb
40  ./network.sh deployCC -ccn fabcar -ccv 1 -cci initLedger -ccl ${CC_SRC_LANGUAGE} -ccp ${CC_SRC_PATH}
41  popd
```

- As you see, **startFabric.sh** calls the **network.sh** script three times, supplying various arguments.
  - **The first call instructs network.sh to shut down the network**. It does not matter if the network is not running; no errors are generated. It always runs and is a safety measure to ensure that a new error-free network is launched.
  - The second execution, or **createChannel** command, **brings up the network, creates a channel (network), and sets the world state database to CouchDB**. After this command completes, the network is up and running.
  - The **third execution deploys the Fabcar chaincode (smart contract) to the network**. The network **provides many default .sh scripts**—for example, **the channel name**, which is ***mychannel***. **Once completed, the Fabcar network is up and running, and our Fabcar smart contract is deployed and initialized**.

Executing startFabric.sh (1)

- Now that we have looked at the **startFabric.sh** script and understand what it does, let's execute it and briefly review the output.
  - It's verbose but informative, so becoming familiar with it will help us better understand what the **network.sh** script is accomplishing for us.
- Open a shell and **make sure Docker is running**. You can check whether Docker is running by executing the following:

```
$ docker -v
Docker version 20.10.10, build b485636
```

- Change to the **fabric-samples/fabcar** directory located where you installed the prerequisites described earlier and execute this:

```
$ docker -v
Docker version 20.10.10, build b485636
```

Executing startFabric.sh (2)

- Your console will begin scrolling text output and continue to completion. The script will take a few minutes to complete.
- Once it's completed, you can save the text output for use and study. It contains the step-by-step sequence used to bring up the network.
- **We are interested in the last command and the sequence of commands used to deploy the Fabcar smart contract**.

Executing startFabric.sh (3)

- Before we look at the smart contract deployment, let's take a quick look at the first part of the output, because **it provides useful information**, including the **channel name** and **versions**:

```
Creating channel 'mychannel'.
If network is not up, starting nodes with CLI timeout of '5' tries and CLI delay of '3' seconds and using database 'couchdb with crypto from '
Certificate Authorities'
Bringing up network
LOCAL_VERSION=2.4.0
DOCKER_IMAGE_VERSION=2.4.0
CA_LOCAL_VERSION=1.5.2
CA_DOCKER_IMAGE_VERSION=1.5.2
```

- We see our channel name is ***mychannel***, and the Fabric version is 2.4.0 (indicated by LOCAL_VERSION) and DOCKER_IMAGE_VERSION.
- The credential authority is version 1.5.2.

Executing startFabric.sh (4)

Generate Fabric Certificate Authorities

- Next, we are informed the **script is going to generate certificates for CAs**, **create the network**, and **create Docker containers** for **organization 1**, **organization 2**, and the **orderer credential authorities**.
- Credential authorities play an important role in managing identities for each organization and the orderer:

```
Generating certificates using Fabric CA
Creating network "fabric_test" with the default driver
Creating ca_orderer ... done
Creating ca_org1    ... done
Creating ca_org2    ... done
Creating Org1 Identities
Enrolling the CA admin
```

- **Then we are informed that the script is going to create the identities for organization 1**. This will take several commands to accomplish. The script repeats these steps for organization 2 and the orderer.

Executing startFabric.sh (5)

Invoke the Peer Chaincode

- We will discuss the smart contract deployment **after we look at the last command executed by the script**, **which is peer chaincode invoke**.
- We **use this last command to initialize the Fabcar smart contract**:

```
invoke fcn call:{"function":"initLedger","Args":[]}
+ peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile /Users/remigijus/HyperFabric/fabric-samples/test-network/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n fabcar --peerAddresses localhost:7051 --tlsRootCertFiles /Users/remigijus/HyperFabric/fabric-samples/test-network/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses localhost:9051 --tlsRootCertFiles /Users/remigijus/HyperFabric/fabric-samples/test-network/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt --isInit -c '{"function":"initLedger","Args":[]}'
+ res=0
2021-12-09 23:39:14.294 EET 0001 INFO [chaincodeCmd] chaincodeInvokeOrQuery -> Chaincode invoke successful. result: status:200
Invoke transaction successful on peer0.org1 peer0.org2 on channel 'mychannel'
```

- The **peer** command has many subcommands, and one we are interested in is **peer chaincode invoke**.
- The **invoke** subcommand can **invoke smart contract functions, which we will do shortly**.

Executing startFabric.sh (6)

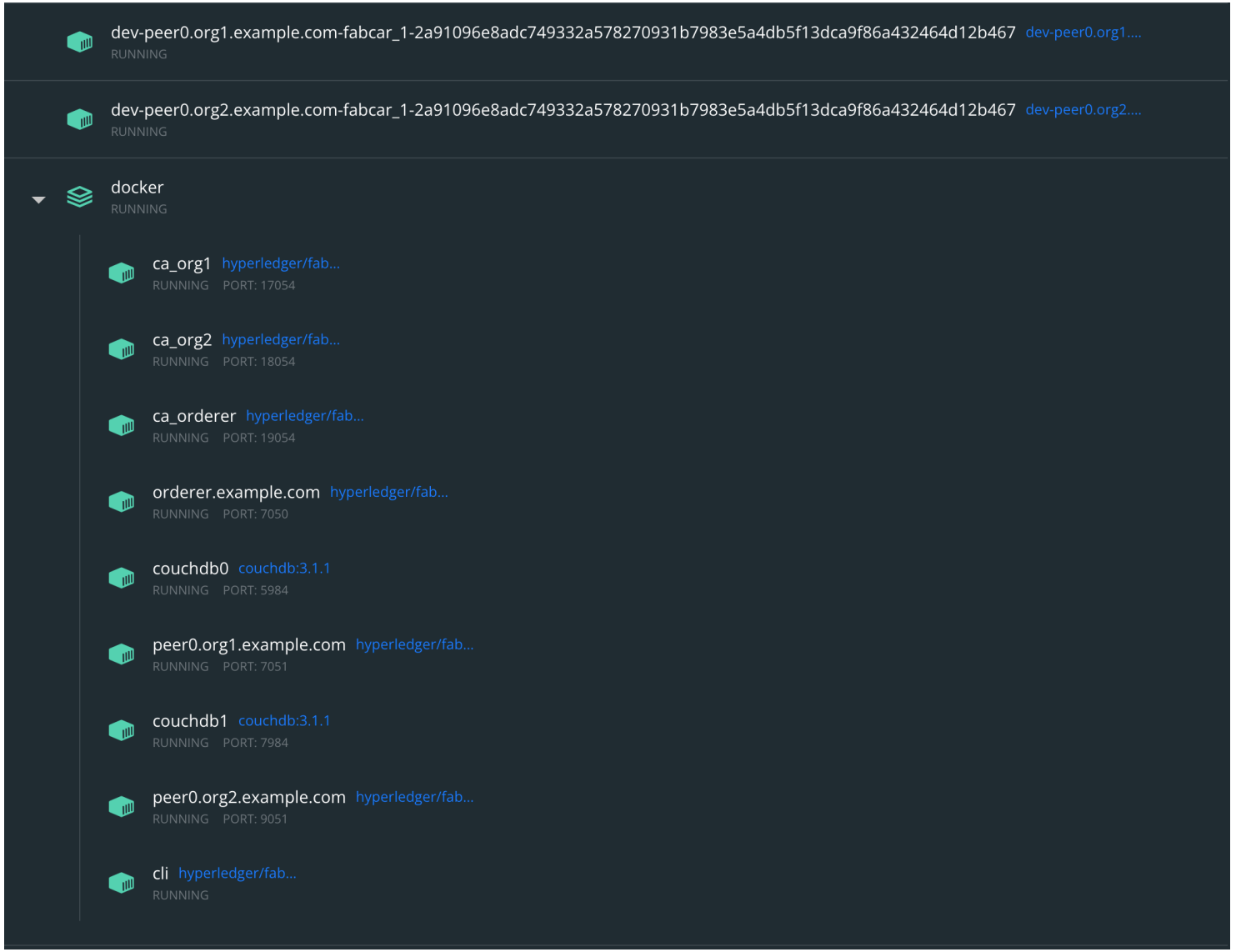
Invoke the Peer Chaincode

```
invoke fcn call:{"function":"initLedger","Args":[]}
+ peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile /Users/remigijus/HyperFabric/fabric-samples/test-network/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n fabcar --peerAddresses localhost:7051 --tlsRootCertFiles /Users/remigijus/HyperFabric/fabric-samples/test-network/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses localhost:9051 --tlsRootCertFiles /Users/remigijus/HyperFabric/fabric-samples/test-network/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt --isInit -c '{"function":"initLedger","Args":[]}'
+ res=0
2021-12-09 23:39:14.294 EET 0001 INFO [chaincodeCmd] chaincodeInvokeOrQuery -> Chaincode invoke successful. result: status:200
Invoke transaction successful on peer0.org1 peer0.org2 on channel 'mychannel'
```

- The **last parameter switch is -c**.
- It is followed by an argument that designates the smart contract function to execute and any optional arguments.
- **We will use this command as one method to invoke a smart contract transaction and evaluate a query**.
- **Our primary purpose for this script is to start up our fully functional Fabric network, so we can develop Fabric smart contracts**.
- The secondary purpose is to capture and leverage the script output.

Test network in the docker Dashboard (1)

- The test network configuration for Fabcar in the Docker Dashboard has the following servers running in Docker containers:



Test network in the docker Dashboard (2)

- ***ca_org2***
  - The credential authority for organization 2
- ***ca_orderer***
  - The credential authority for the network orderer
- ***ca_org1***
  - The credential authority for organization 1
- ***couchdb1***
  - The world state database for organization 2
- ***orderer.example.com***
  - The network orderer that creates blocks and sends them for commit
- ***couchdb0***
  - The world state database for organization 1

Test network in the docker Dashboard (3)

- ***peer0.org2.example.com***
  - *The peer for organization 2*
- ***peer0.org1.example.com***
  - *The peer for organization 1*

- The two top containers are the smart contract runtime containers for each organization's peer.
- The **world state database** for the organizations in our test network is **CouchDB**.
- We can access CouchDB with our browser by using the Fauxton web application, which is part of CouchDB (will do this soon)

Invoking Smart Contract Transactions

Vilnius University

Invoking smart contract transactions

1. The **peer chaincode invoke** command is the first method **we will look at and execute to invoke our Fabcar smart contract functions**.
2. The **next method** we will look at and use **incorporates the Hyperledger Fabric SDK for Node.js**, which can be used for **developing smart contract clients**.
   - By **using the Hyperledger Fabric SDK for Node.js**, you can develop both **command line Fabric smart contract clients**, like Fabcar, and **Fabric smart contract clients that incorporate a UI into the design of the Fabric smart contract client application**

Peer command (1)

- The **peer** command is a **Hyperledger Fabric core binary**.
- When you installed the prerequisites, the **peer** binary **along with several other binaries** were downloaded and placed into the **bin/** subdirectory of **fabric-samples/**.
- This binary has five functions, or subcommands, it performs:

  peer chaincode [option] [flags]
  peer channel [option] [flags]
  peer node [option] [flags]
  peer version [option] [flags]
- The peer subcommand we are interested in is the chaincode subcommand.
- This subcommand also has several subcommands: install, instantiate, invoke, list, package, query, signpackage, upgrade

Peer command (2)

- We want to use the invoke **subcommand**.
- Some other peer chaincode subcommands **were used in the script that launches the test network**.
- Using the peer command involves many parameters, so in a practice environment, we use variables to minimize the length of the resulting peer command-line text.
- **We will use environment variables** so we can execute peer chaincode invoke against our Fabcar test network.

Peer command (3)

- When we executed startFabric.sh, we were in the **fabric-samples/fabcar** subdirectory.
- The script executes a change directory command while saving the current directory, which is **fabric-samples/fabcar**.
- The script changes to **fabric-samples/test-network**. This makes **fabric-samples/test-network** the current directory for the execution of the script.
- **This is important to us** because we are going to reuse the **peer chaincode invoke command from the script to execute our invoke command**.
- For this to work, we need to open a shell and **change the directory to fabric-samples/test-network**.

Peer command (4)

- **Then execute these export commands to configure your environment to**
  - **find the peer binary** in the **bin/ subdirectory**,
  - the created **test-network** configuration in **fabricsamples/config**,
  - and **additional information peer needs to find in order to execute correctly**

```
export PATH=${PWD}/../bin:$PATH
export FABRIC_CFG_PATH=$PWD/../config/
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
```

- Once you execute these, **your environment is configured to execute peer**. To test this, execute the following:

```
$ peer version
peer:
 Version: 2.4.0
```

Peer chaincode invoke read transaction (1)

- Locate the **peer chaincode invoke** command that initializes the Fabcar smart contract and copy it to your editor, so we can refactor it.
- Remember, it is at **the last command executed in the launch script** output and looks like this, **except your paths will be different**:

```
invoke fcn call:{"function":"initLedger","Args":[]}
+ peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile /Users/remigijus/HyperFabric/fabric-samples/test-network/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n fabcar --peerAddresses localhost:7051 --tlsRootCertFiles /Users/remigijus/HyperFabric/fabric-samples/test-network/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses localhost:9051 --tlsRootCertFiles /Users/remigijus/HyperFabric/fabric-samples/test-network/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt --isInit -c '{"function":"initLedger","Args":[]}'
+ res=0
2021-12-09 23:39:14.294 EET 0001 INFO [chaincodeCmd] chaincodeInvokeOrQuery -> Chaincode invoke successful. result: status:200
Invoke transaction successful on peer0.org1 peer0.org2 on channel 'mychannel'
```

- We want to modify the text at the end of the command:

  --isInit -c '{"function":"initLedger","Args":[]}‘

- Into this: -c '{"Args":["queryAllCars"]}‘

Peer chaincode invoke read transaction (2)

- Make sure you delete the --isInit switch.
- **Since we removed the command to be executed for initialization, it will fail and error, because this switch is informing the peer to perform an initialization**.
- Once you modify the command text, it should look like this:

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile
/Users/remigijus/HyperFabric/fabric-samples/test-
network/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-
cert.pem -C mychannel -n fabcar --peerAddresses localhost:7051 --tlsRootCertFiles /Users/remigijus/HyperFabric/fabric-
samples/test-network/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --
peerAddresses localhost:9051 --tlsRootCertFiles /Users/remigijus/HyperFabric/fabric-samples/test-
network/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{"Args":["queryAllCars"]}'
```

- Now copy the command to your shell and execute it.

Peer chaincode invoke read transaction (3)

- The output will look like this:

```
2021-12-10 01:08:00.513 EET 0001 INFO [chaincodeCmd] chaincodeInvokeOrQuery -> Chaincode invoke successful. result: status:200 payload:"[{\"Key\":\"CAR0\",\"Record\":{\"color\":\"blue\",\"docType\":\"car\",\"make\":\"Toyota\",\"model\":\"Prius\",\"owner\":\"Tomoko\"}},{\"Key\":\"CAR1\",\"Record\":{\"color\":\"red\",\"docType\":\"car\",\"make\":\"Ford\",\"model\":\"Mustang\",\"owner\":\"Brad\"}},{\"Key\":\"CAR2\",\"Record\":{\"color\":\"green\",\"docType\":\"car\",\"make\":\"Hyundai\",\"model\":\"Tucson\",\"owner\":\"Jin Soo\"}},{\"Key\":\"CAR3\",\"Record\":{\"color\":\"yellow\",\"docType\":\"car\",\"make\":\"Volkswagen\",\"model\":\"Passat\",\"owner\":\"Max\"}},{\"Key\":\"CAR4\",\"Record\":{\"color\":\"black\",\"docType\":\"car\",\"make\":\"Tesla\",\"model\":\"S\",\"owner\":\"Adriana\"}},{\"Key\":\"CAR5\",\"Record\":{\"color\":\"purple\",\"docType\":\"car\",\"make\":\"Peugeot\",\"model\":\"205\",\"owner\":\"Michel\"}},{\"Key\":\"CAR6\",\"Record\":{\"color\":\"white\",\"docType\":\"car\",\"make\":\"Chery\",\"model\":\"S22L\",\"owner\":\"Aarav\"}},{\"Key\":\"CAR7\",\"Record\":{\"color\":\"violet\",\"docType\":\"car\",\"make\":\"Fiat\",\"model\":\"Punto\",\"owner\":\"Pari\"}},{\"Key\":\"CAR8\",\"Record\":{\"color\":\"indigo\",\"docType\":\"car\",\"make\":\"Tata\",\"model\":\"Nano\",\"owner\":\"Valeria\"}},{\"Key\":\"CAR9\",\"Record\":{\"color\":\"brown\",\"docType\":\"car\",\"make\":\"Holden\",\"model\":\"Barina\",\"owner\":\"Shotaro\"}}]"
```

- We just **invoked the Fabcar smart contract**.
- What **we invoked** was **a query**, **which is a read transaction**.
- **When we invoke, we execute either a write or a read transaction**.
- The **read transactions are not committed to the ledger**; **only write transactions are committed**, provided they get properly endorsed and validated.
- The Fabcar smart contract transaction we invoked was the **queryAllCars** transaction

Peer chaincode invoke write transaction (1)

- Now let's use peer chaincode invoke to invoke a Fabcar **write transaction**.
- We can use the same command we used for the query. All we need to do is change the argument to the -c switch.
- Let's execute **changeCarOwner**:

  -c '{"Args":["changeCarOwner", "CAR0", "Mark"]}'

- After you change the -c switch argument, you should have a command:

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile
/Users/remigijus/HyperFabric/fabric-samples/test-
network/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-
cert.pem -C mychannel -n fabcar --peerAddresses localhost:7051 --tlsRootCertFiles /Users/remigijus/HyperFabric/fabric-
samples/test-network/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --
peerAddresses localhost:9051 --tlsRootCertFiles /Users/remigijus/HyperFabric/fabric-samples/test-
network/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c
'{"Args":["changeCarOwner", "CAR0", "Mark"]}'
```

Peer chaincode invoke write transaction (2)

- Now execute it and see this for the result:

```
2021-12-10 01:19:49.509 EET 0001 INFO [chaincodeCmd] chaincodeInvokeOrQuery -> Chaincode invoke successful. result: status:200
```

- **This shows the transaction executed successfully!**
- Since we change the owner of CAR0, let's check it with another query, but this time for specific data.
- Again using the same **peer** command, we just need to change the -c switch argument: -c '{"Args":["queryCar", "CAR0"]}'

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile
/Users/remigijus/HyperFabric/fabric-samples/test-
network/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-
cert.pem -C mychannel -n fabcar --peerAddresses localhost:7051 --tlsRootCertFiles /Users/remigijus/HyperFabric/fabric-
samples/test-network/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --
peerAddresses localhost:9051 --tlsRootCertFiles /Users/remigijus/HyperFabric/fabric-samples/test-
network/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{"Args":["queryCar",
"CAR0"]}'
```

Peer chaincode invoke write transaction (3)

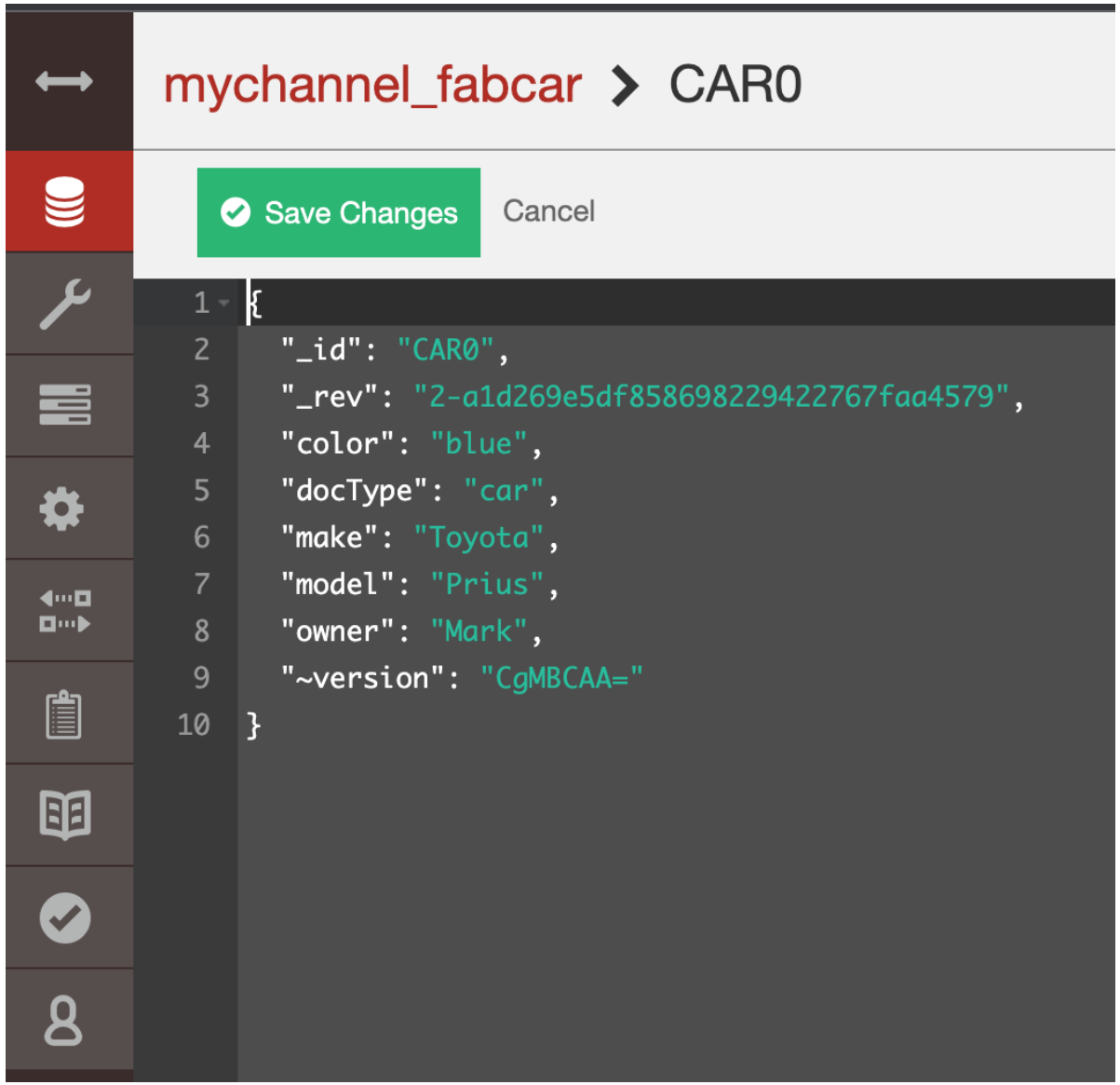
- After executing, you should see this:

```
2021-12-10 01:27:19.879 EET 0001 INFO [chaincodeCmd] chaincodeInvokeOrQuery -> Chaincode invoke successful. result: status:200 payload:"{\"col
or\":\"blue\",\"docType\":\"car\",\"make\":\"Toyota\",\"model\":\"Prius\",\"owner\":\"Mark\"}"
```

- It worked! We changed the owner.
- The key for our write transaction, **changeCarOwner**, is **CAR0**.
- Remember, the ledger has two parts: the world state and the blockchain ledger (a file-based implementation of an immutable linked list, in simple terms).
- The world state is the current state of the blockchain for a given key.
- Our world state is implemented by CouchDB for our Fabcar test network
- We can see our change by using the CouchDB browser interface

View our change using CouchDB browser interface

- Open your browser: http://127.0.0.1:5984/_utils/#login
- and log in to CouchDB with:
  - the **username admin** and **password adminpw**
- Then go to this URL in CouchDB:
- http://127.0.0.1:5984/_utils/#database/mychannel_fabcar/CAR0
- You should see that the **owner of CAR0 is Mark**, as displayed in Figure



Fabric SDK for Node.js Command-Line Application

Fabcar smart contract client

Vilnius University

Using Fabcar smart contract client (1)

- Now we are going to examine the **Fabcar smart contract client** and **use it to invoke the contract**.
- The **Fabcar smart contract client**, as discussed earlier in this lecture, is a group of four Node.js JavaScript command-line applications: enrollAdmin.js, invoke.js, query.js, and registerUser.js.
- We will execute each one and look at how they work to invoke the Fabcar smart contract.

```
~/HyperFabric/fabric-samples/fabcar/javascript on  main  22:55:59
$ ll *.js
-rw-r--r--  1 remigijus  staff   2.1K Dec  9 13:11 enrollAdmin.js
-rw-r--r--  1 remigijus  staff   2.1K Dec  9 13:11 invoke.js
-rw-r--r--  1 remigijus  staff   2.1K Dec  9 13:11 query.js
-rw-r--r--  1 remigijus  staff   2.8K Dec  9 13:11 registerUser.js
```

Using Fabcar smart contract client (2)

Installing dependencies

- To begin, **open a shell and change to directory:** fabric-samples/fabcar/javascript/
- The **Fabcar client is a Node.js command-line application**, so we need to execute the following:

```
$ npm install
```

- This command will install project dependencies into our Fabcar project.
- When the command completes, you should have a **node_modules/** subdirectory.

```
~/HyperFabric/fabric-samples/fabcar/javascript on  main  17:20:51
$ ll
total 576
-rw-r--r--    1 remigijus  staff   2.1K Dec  9 13:11 enrollAdmin.js
-rw-r--r--    1 remigijus  staff   2.1K Dec  9 13:11 invoke.js
drwxr-xr-x  266 remigijus  staff   8.3K Dec 13 17:20 node_modules
-rw-r--r--    1 remigijus  staff   266K Dec 13 17:20 package-lock.json
-rw-r--r--    1 remigijus  staff   1.0K Dec  9 13:11 package.json
-rw-r--r--    1 remigijus  staff   2.1K Dec  9 13:11 query.js
-rw-r--r--    1 remigijus  staff   2.8K Dec  9 13:11 registerUser.js
drwxr-xr-x    3 remigijus  staff    96B Dec  9 13:11 wallet
```

Using Fabcar smart contract client (3)

- If we look in the **node_modules** subdirectory for installed Fabric modules, among others we will find the Fabric modules listed here:
  - **fabric-ca-client** - is used by the Fabcar client, but is not required for a Fabric smart contract client. You need to import and use **fabric-ca-client** only if your smart contract client needs to interact with Fabric CA to manage the life cycle of user certificates. This includes the ability to enroll, register, renew, and revoke users. If your smart contract client does not perform any of these functions, you do not need fabric-ca-client
  - **fabric-common** - contains code used by both the fabric-ca-client and fabric-network modules.
  - **fabric-network** - is the only Fabric module required for a Fabric smart contract client. It connects your client to the network and provides the ability to invoke transactions, including writing to the ledger and querying the world state and ledger.
  - **fabric-protos** - implements the Protocol Buffers protocol, which is a binary encoded communication protocol
- These modules, which represent the Hyperledger Fabric SDK for Node.js, can be found in the Fabric GitHub repo fabric-sdk-node

Using Fabcar smart contract client (4)

- Now that our dependencies are installed, we can perform the following:
  - Enroll our application administrator.
  - Register our application user.
  - Invoke a write transaction.
  - Query the world state.
- **We first need to enroll an application admin to create an admin-level wallet**.
- Then we can register users in an application with an assigned user wallet to secure network access.

Using Fabcar smart contract client (5)
Enroll our application administrator and register a user

- To enroll our administrator, we execute the **enrollAdmin.js** application and you should see the following result:

```
~/HyperFabric/fabric-samples/fabcar/javascript on  main  17:41:01
$ node ./enrollAdmin.js
Wallet path: /Users/remigijus/HyperFabric/fabric-samples/fabcar/javascript/wallet
Successfully enrolled admin user "admin" and imported it into the wallet
```

- Now we can execute **registerUser.js** to register our user an you shoud see:

```
~/HyperFabric/fabric-samples/fabcar/javascript on  main  17:41:17
$ node ./registerUser.js
Wallet path: /Users/remigijus/HyperFabric/fabric-samples/fabcar/javascript/wallet
Successfully registered and enrolled admin user "appUser" and imported it into the wallet
```

- Great! You should see a file called **admin.id** in the **wallet/ subdirectory**

Using Fabcar smart contract client (6)
Invoke a write transaction

- We can now invoke our smart contract by using the user identity we just registered.
- Let's execute **invoke.js** and you should see the following result:

```
~/HyperFabric/fabric-samples/fabcar/javascript on  main  21:44:26
$ node ./invoke.js
Wallet path: /Users/remigijus/HyperFabric/fabric-samples/fabcar/javascript/wallet
Transaction has been submitted
```

- Let's look at the **invoke.js** code

Using Fabcar smart contract client (7)
invoke.js file analysis (1)

- The application imports from only fabric-network (Line 9):

```
 9   const { Gateway, Wallets } = require('fabric-network');
10   const fs = require('fs');
11   const path = require('path');
12
13   async function main() {
```

- This means it requires no interaction with the CA to invoke transactions.
- It also makes use of **Wallets** from fabric-network.
  - This makes sense because we need an identity to submit requests to the network, and identities for a client are stored in a wallet.
- Then it has a **main** function (Line 13), and all logic is contained **within the main**.

Using Fabcar smart contract client (8)
invoke.js file analysis (2)

- Then it **load the network configuration** (Lines 16-17), which contains the information for connecting to the network, **creating the wallet** (Lines 19-22), and **checking for an identity** (Lines 25-29).
  - We need the user identity to exist so the gateway can find it; if it does not we **return**.

```
15      // load the network configuration
16      const ccpPath = path.resolve(__dirname, '..', '..', 'test-network', 'organizations', 'peerOrganizations', 'org1.e
17      let ccp = JSON.parse(fs.readFileSync(ccpPath, 'utf8'));
18
19      // Create a new file system based wallet for managing identities.
20      const walletPath = path.join(process.cwd(), 'wallet');
21      const wallet = await Wallets.newFileSystemWallet(walletPath);
22      console.log(`Wallet path: ${walletPath}`);
23
24      // Check to see if we've already enrolled the user.
25      const identity = await wallet.get('appUser');
26      if (!identity) {
27          console.log('An identity for the user "appUser" does not exist in the wallet');
28          console.log('Run the registerUser.js application before retrying');
29          return;
30      }
```

Using Fabcar smart contract client (9)
invoke.js file analysis (3)

- To connect to the network, we use the **Gateway** we imported from fabric-network to create a gateway object and use it to connect (Line 33).
  - The arguments we pass to the connect function are the connection information, the wallet, the name of the identity to use, and some discovery options used to locate peers (Line 34)
- After connecting, we use the gateway to get the network, which is our channel identified by our channel name (mychannel)
- With the network object, we get a connection to our Fabcar smart contract (L. 40)

```
32        // Create a new gateway for connecting to our peer node.
33        const gateway = new Gateway();
34        await gateway.connect(ccp, { wallet, identity: 'appUser', discovery: { enabled: true, asLocalhost: true } });
35
36        // Get the network (channel) our contract is deployed to.
37        const network = await gateway.getNetwork('mychannel');
38
39        // Get the contract from the network.
40        const contract = network.getContract('fabcar');
```

Using Fabcar smart contract client (10)

invoke.js file analysis (4)

- **Finally, we get to invoke our smart contract.**
- We are performing a transaction that will change the ledger, so this is a **write transaction**, and we expect it to be **committed**.
- The **contract API** function we want to use is submitTransaction.
  - This will call our **createCar smart contract function** with the arguments shown in Figure
- As you can see, we invoke our smart contract indirectly through the contract API.
  - This design pattern is powerful but also risky if mitigations are not implemented to prevent unwanted side effects from unexpected character or binary data. The best industry practices should be implemented to validate and protect data integrity.

```
42          // Submit the specified transaction.
43          // createCar transaction - requires 5 argument, ex: ('createCar', 'CAR12', 'Honda', 'Accord', 'Black', 'Tom')
44          // changeCarOwner transaction - requires 2 args , ex: ('changeCarOwner', 'CAR12', 'Dave')
45          await contract.submitTransaction('createCar', 'CAR12', 'Honda', 'Accord', 'Black', 'Tom');
46          console.log('Transaction has been submitted');
```

Using Fabcar smart contract client (11)
query the world state

- Let's execute **query.js** and you should see the following result:

```
~/HyperFabric/fabric-samples/fabcar/javascript on main 21:44:34
$ node ./query.js                                                                                    ‹system›
Wallet path: /Users/remigijus/HyperFabric/fabric-samples/fabcar/javascript/wallet
Transaction has been evaluated, result is: [{"Key":"CAR0","Record":{"make":"Toyota","model":"Prius","colour":"blue","owner":"Mark"}},{"Key":"
CAR1","Record":{"make":"Ford","model":"Mustang","colour":"red","owner":"Brad"}},{"Key":"CAR12","Record":{"make":"Honda","model":"Accord","col
our":"Black","owner":"Tom"}},{"Key":"CAR2","Record":{"make":"Hyundai","model":"Tucson","colour":"green","owner":"Jin Soo"}},{"Key":"CAR3","Re
cord":{"make":"Volkswagen","model":"Passat","colour":"yellow","owner":"Max"}},{"Key":"CAR4","Record":{"make":"Tesla","model":"S","colour":"bl
ack","owner":"Adriana"}},{"Key":"CAR5","Record":{"make":"Peugeot","model":"205","colour":"purple","owner":"Michel"}},{"Key":"CAR6","Record":{
"make":"Chery","model":"S22L","colour":"white","owner":"Aarav"}},{"Key":"CAR7","Record":{"make":"Fiat","model":"Punto","colour":"violet","own
er":"Pari"}},{"Key":"CAR8","Record":{"make":"Tata","model":"Nano","colour":"indigo","owner":"Valeria"}},{"Key":"CAR9","Record":{"make":"Holde
n","model":"Barina","colour":"brown","owner":"Shotaro"}}]
```

- **Can you find the car we created, CAR12?**
- It's the third one right after CAR1.
- Great, our **submitTransaction** was committed.

Using Fabcar smart contract client (12)

- If we looked in **CouchDB**, we would see this data in our world state database: mychannel_fabcar.

id	key	value
initialized	initialized	{ "rev": "1-9a400e5a0ed9adb2b496f76cdc5ab946" }
CAR0	CAR0	{ "rev": "2-ee94c07f210e563c8d0a2e3a8967dbae" }
CAR1	CAR1	{ "rev": "1-01bbe2ff3517d10fd699882b2960d677" }
CAR12	CAR12	{ "rev": "1-c162e08e40459dd836ca6b32628346d1" }
CAR2	CAR2	{ "rev": "1-237369c02bf4e7162b0e0169cae12a8a" }
CAR3	CAR3	{ "rev": "1-f53e7a6da7e7d6dcf41743b4c578f9ab" }
CAR4	CAR4	{ "rev": "1-7aee4831a389d89791037d8b14750922" }
CAR5	CAR5	{ "rev": "1-7e9e2c98e5541b7809b33d4770ff57fb" }
CAR6	CAR6	{ "rev": "1-b0a10ecb101397c4c863e23da8e318ff" }
CAR7	CAR7	{ "rev": "1-c5f39852c4c0f51de0fcb3b5611a9c0f" }
CAR8	CAR8	{ "rev": "1-85440d3604bfa3140b891ef34191578c" }
CAR9	CAR9	{ "rev": "1-687941635d74b3e269aa8ac63a311671" }

Using Fabcar smart contract client (13)
Summary

- We have executed 4 applications that make up the Fabcar smart contract client.
- We saw the differences and common code they use. Each performs one application function, and has dependencies that determine the order they are executed.
- You learned we need to
  - first enroll an application administrator,
  - and then use the administrator to register and enroll application users.
  - Once we have a user identity, we can submit transactions to commit data to the ledger
  - and query the world state.
- These applications serve as good examples for solutions that are best implemented with a command-line application, like batch jobs or serverless commands.

Lecture summary

- In this lecture, you learned how to invoke a Fabric smart contract.
  - We began with a review of the Hyperledger **fabric-samples**, **Fabcar smart contract**, and **Fabric test network**.
- We then launched our Fabric test network and discussed the launch script and its output to gain an understanding of what it takes to stand up a Fabric network for developing Fabric smart contracts.
  - Besides launching the test network, the script also deployed the Fabcar smart contract. We went through the steps used by the script to deploy a smart contract, which will help you to understand and perform the deployment task for your own smart contracts.
- Next, we executed each of the four Fabcar client's Node.js command-line applications that together make up the Fabcar smart contract client.
  - The Fabcar applications use the SDK to call the Fabric APIs for interacting with Fabric smart contracts, credential authorities, and wallets as well as for invoking smart contracts.

Vilnius University

AČIŪ UŽ DĖMESĮ!

KONTAKTAI

Prof. dr. Remigijus Paulavičius,
remigijus.paulavicius@mif.vu.lt

Dr. Ernestas Filatovas:
ernestas.filatovas@mif.vu.lt