**CSE598 Engineering Blockchain Applications**

**Project 1: Hyperledger Fabric Private Blockchain and Smart Contracts.**

This document contains description of tasks required to complete the PROJECT 1 assignment. This project will help you familiarize with a private blockchain ecosystem by understanding, examining, writing and executing smart contracts for a simple-use case of product records management.

For this project you will be working with the Hyperledger Fabric blockchain framework. Hyperledger is an opensource community focused on developing a suite of stable frameworks, tools and libraries for enterprise-grade blockchain deployments. Hyperledger was established under the Linux Foundation. It serves as a neutral home for various distributed ledger frameworks including Hyperledger Fabric. Hyperledger Fabric is an open source enterprise-grade permissioned distributed ledger technology (DLT) platform, designed for use in enterprise contexts, that delivers some key differentiating capabilities over other popular distributed ledger or blockchain platforms. Read more about Hyperledger Fabric on this link ( <https://hyperledger-fabric.readthedocs.io/en/release-1.4/whatis.html>).

**Smart Contract**

Smart contracts are mediums that are used to manage digital assets, store information, make decisions, and interact with other smart contracts. Hyperledger Fabric smart contracts usually manipulate JSON-like digital assets (arrays of key-value pairs) of any complexity. For every digital asset we want to store on a Hyperledger Fabric blockchain, there must be a smart contract in place for its management (writing data on the blockchain, updating, reading, etc.).

In Hyperledger Fabric smart contracts are packaged into *chaincodes* and the *chaincode* is deployed on the Hyperledger Fabric blockchain. Chaincode is a term local to the Hyperledger Fabric framework and, for now, you can think of chaincode and smart contract as synonyms. To read more about chaincodes in Hyperledger Fabric, visit the link (<https://hyperledger-fabric.readthedocs.io/en/release-1.4/chaincode.html>).

Writing smart contacts on Hyperledger Fabric network requires three classes: State class, Contract Class, Statelist class.

* State class: Used to represent the asset on which the smart contract will be applied
* Contract class: Used to define methods that are setup in the contract
* Statelist class: Used to interact with the blockchain network. In this class we define methods to add, get and update data on the blockchain network.

In this project, you will be writing a smart contract for supply chain management that manages assets of home appliances company. For this project, you will receive a codebase that needs to be completed so that the contract that is fully functional. You will be in charge of complementing several smart contract functions.

1. A function that creates product assets on the Hyper ledger Fabric blockchain network.
2. A function that updates one attribute of a product record.
3. Several functions that allow reading/ accessing the information about the company’s products using Couch DB-enabled data indexing and querying.

In this project you are provided with semi-written project code (codebase). In some code blocks, you will find the following text: *“GRADED FUNCTION: Function Name”*. These functions are half-done: you will notice that they are missing some key lines of code for them to work when deployed. Your overall task is to update and finish the code for these functions and submit you work to the Grading Service.

Once again, the use-case for this project is supply chain management of a home appliance company. The smart contract must be able to write product records on the blockchain and execute various queries based on attributes of product records. Provided code base includes three files:

1. ProductRecord.js
2. ProductContract.js
3. ProductList.js

Please examine the **ProductRecord.js** file that defines how a product record looks like. ProductRecord consists of the following attributes:

* Product\_ID- text that uniquely identifies a product
* Name – Product name
* Mfg\_Date - date of manufacture
* Product\_Type [Product type can be TV, Refrigerator, Oven, Washing Machine]

In the provided code base, specifically the productcontract.js file containing the code of our smart contract, you will notice that every function has one regular input parameter: **ctx**. This parameter refers to the transaction context within the execution of the function. At the beginning of the ProductContract file, you can see the definition of the **ProductContext** and its constructor, where **ProductList** class is initialized. This is done so that the **ProductList** object could be referenced in every function without initializing it every time, but simply calling **ctx. productList.[function]**. Inside the ProductContract class there is a default createContext function that makes the connection between the smart contract and the context, enabling us to use code like **ctx. productList.[function]**. You can read more about the transaction context, motivation and use-cases behind it on this link[[1]](#footnote-1). **CTX IS ALWAYS THE FIRST PARAMETER IN A FUNCTION – DO NOT REMOVE IT.**

Now that you are familiar with the concepts behind Hyperledger Fabric and smart contracts, that we have examined the use-case behind this project, established what type of digital record our smart contract is supposed to manage (and what attributes it holds), continue with executing the following tasks.

**Task 1 – Complete the getProductByKey function (ProductContract.js)**

Instructions

The getProductByKey function receives product\_ID and name. These two attributes together make a composite key for this record, and by this key the record is searchable in the blockchain. The first line of this function creates such a key for you to use in the rest of the function. Complete the code of the getProductByKey function so that it returns the record of the product for the created composite key. Use a function from the ProductList class, which can be referenced from inside the getProductByKey by calling ctx.productList.[functionName], such as ctx.productList.addPRecord.

**Task 2 – Complete the getter and setter methods for quantity field of the Product. (ProductRecord.js)**

Instructions

In this task you need to write code to add a new attribute to a product record, the quantity attribute representing the number of items of the product currently available with the supplier. Complete the **setQuantity** function to set the quantity field of the productRecord. This function takes an integer input and assigns it to the quantity field. Complete the **getQuantity** function to return the quantity field of the ProductRecord.

**Task 3 – Complete the updateQuantity function (ProductContract.js)**

Instructions

The **updateQuantity** function receives the transaction context, product\_ID, name and quantity. To update the product’s current quantity first retrieve the product record by calling the ctx.productList.getPRecord. The **getPRecord function** receives the composite key as an input parameter (the key is made from the username and name fields). Update the quantity on the ProductRecord using the function implemented in task 2. Update the ProductRecord on the ledger by calling ctx.productList.[functionName]

**Task 4 – 6** **PREREQUISITES (CouchDB-enabled indexed querying)**

For tasks 4-6 you will need to perform a crucial preparatory step, otherwise the tasks are not going to be graded – you have to physically build the index files on the top of the attributes that are defined in the **Product** class. For writing indexes follow the path: *Project\_foldername>META-INF>statedb>couchdb>indexes.* Inside this folder you will find one file *ProductTypeIndex.json*. This folder structure and this file are telling the Hyperledger Fabric framework that, when deploying the smart contract, you also want to create indexes for certain attribute names for the records your smart contract is going to manage. Indexes enable the CouchDB to perform faster searches on all records given a certain query string. Our smart contract manages product records, so we want to index two fields: (1) the product\_type and (2) mfg\_date. As the producttype*Index.json* already exist, create the **mfg\_dateIndex.json** file in the indexes folder with the same structure as *prodcuttypeIndex.json*, but referencing another attribute

**Task 4 – Complete queryByProductType function (ProductContract.js).**

Instructions

This function takes transaction context and product\_type as input. Your task is to construct the JSON CouchDB selector query-string object (*queryString*) that uses the ProducttypeIndex. See what CouchDB selector queries are and a few examples on this link[[2]](#footnote-2). To make sure the query will actually use the index that you have created you must specify the *use*\_*index* attribute inside the *queryString*. Once the *queryString* is build, pass it to the *queryWithQueryString*. This function will return a list of records that correspond to the product\_type that is passed, and you need to return this list from the **queryByProductType** function.

**Task 5 – Complete querybyMfgdate function (ProductContract.js).**

Instructions

This function takes transaction context and mfg\_date as input. Your task is to construct the JSON CouchDB selector query-string object(*queryString*) that uses the mfgdateIndex. To make sure the query will use the index you have created you must specify the *use\_index* attribute inside the *queryString.* Once the *queryString* is build, pass it to the *queryWithQueryString.* This function will return a list of records that correspond to the mfg\_date that is passed as the input and you need to return this list from the **querybyMfgdate** function.

**Task 6 – Complete querybyProduct\_Type\_Dual function (ProductContract.js).**

Instructions

This function takes the transaction context and 2 product types as input. Your task is to construct the JSON CouchDB selector query-string object that uses the 2 product\_type indexes. To make sure the query will actually use the index that you have created you must specify the *use*\_*index* attribute inside the *queryString*. Once the *queryString* is build, pass it to the *queryWithQueryString*. This function will return a list of records that correspond to the product\_type that is passed, and you need to return this list from the **queryByProductType** function.

Use the helper functions of the ProductContract class to query the query the database and return list of records with the given product\_types.

**Task 7 – Complete the unknownTransaction function**

Instructions

In smart contracts, it is possible to get a function’s name wrong when calling the contract. In this case, the smart contract usually returns an error. A good practice is to implement a certain default function that will, instead, execute every time a function is invoked that does not exist in the smart contract. This default function is called *unknownTransaction* and it receives the transaction context only.

The purpose of this function is to throw an error when a function called doesn’t exist in the contract. Complete the function to return a string message [“Function Name Missing”]

To read more about unknown transaction refer:

<https://hyperledger.github.io/fabric-chaincode-node/master/api/>

1. <https://hyperledger-fabric.readthedocs.io/en/release-2.2/developapps/transactioncontext.html> [↑](#footnote-ref-1)
2. <https://docs.couchdb.org/en/stable/api/database/find.html#selector-syntax> [↑](#footnote-ref-2)