UPMopoly: Implementation in Hyperledger

Benedek Kiraly, Moritz Walther

EIT Digital Masters in FinTech

January 2021

Introduction

In the course “Blockchain and Services for Fintech Enterprise Integration” we have examined various enterprise integration technologies. Enterprise integration refers to a field focusing on the interconnection of different systems and how data is exchanged between them. While the term historically referred to sharing data between systems within the scope of one company, changing models of innovation from closed to open approaches have meant the technology has taken on a broader scope. Enterprise integration has recently become one of the most important topics affecting the financial services industry. In the EU this trend driven largely by the Revised Payment Services Directive (PSDII). Banks and other payment service providers are now required to share data between each other. Many used this regulatory imperative to consider new opportunities in “open banking” - sharing information beyond the regulatory imperative, creating ecosystems, redesigning their internal integration approaches and exploring new technologies. The course walked us through data security practices and cryptography as an essential prerequisite for sharing data securely. We explored different integration architectures especially Service Oriented Architectures. We were introduced to enterprise integration technologies especially REST. Finally, we had special focus on blockchain technologies as a method of enterprise integration for trustless environments. The course culminated in implementing a game “UPMopoly” using Hyperledger infrastructure and a smart contract. UPMopoly is a simplified version of the board game Monopoly where players must buy, sell and pay rent on faculties (as opposed to properties). The blockchain system is used to account for ownership and keep track of finances.

Participants

There is only one kind of participant in the game namely the players. Players are defined in the Player.java file and in the corresponding Player class. Players can have two states: playing or eliminated. Players have the properties playerNumber, name and initialAmount. The initial amount defines the starting sum of money that they have to spend on buying and selling properties, or paying rent. If the money is spent the players state must change to eliminated. The class contains getters and setters for each of these properties. It also contains a toString method. It has a serialize and deserialize method for converting between JSON objects and Java objects. There is a factory method called createInstance for instantiating a player object. There is also an implied key property. The Hyperledger ledger architecture contains both a blockchain and a world state database. The blockchain is of course the immutable transaction log. The world state is a separate construct. It is a database storing current values of the ledger states – derived from the blockchain and making it significantly easier to access the current state without having to calculate this from the ledger. These states are expressed in key-value pairs by default. Here the key is such a pair and is defined in another method and may be passed to the object.

Assets

The Asset which the contract governs are the faculties. Faculties within UPMopoly can be logically equated with the familiar concept of properties within the board game Monopoly. Interactions with assets change the players balances of the virtual in game currency. The Faculty class is defined in Faculty.java. Faculty objects are instantiated by the smart contract. These objects have the properties ID, name, salePrice and rentalFee. The faculties can have the states bought or free indicating whether they are owned by a player. Just like the Player class the Faculty class contains getters and setters for each of these properties. It also contains a toString method. It has a serialize and deserialize method for converting between JSON objects and Java objects.

Transactions

The game has three kinds of transactions. These transactions are specifically invoke transactions (as opposed to deploy transactions) as per the Hyperledger architecture. They refer to the deployed chain code and are used to modify the state of the chain as well as return an output of the function. The three kinds of transactions are buy, sell and payRent.

The buy transaction is used for a player to purchase a free faculty. It is implemented in the GameContract.java smart contract in the buyFaculty method. The transaction needs to change the state of the faculty object from free to bought, reduce the virtual currency of the purchasing player and assign ownership. For further details see the Smart Contract section.

The sell transaction is used for a player to give up their ownership of a faculty, and another player to gain ownership. It is implemented in the GameContract.java smart contract in the facultySale method. The transaction needs to change the owner of the property. It does not need to change the state. It reduces the virtual currency balance of the purchasing player and increases that of the seller. For further details see the Smart Contract section.

The final kind of transaction which needs to be implemented is paying rental fees. In the game when a player lands on faculty they need to pay a fee. The ledger must record this. The transaction changes the virtual currency balances of the players. The faculty owners balance is increased while the renting players balance is decreased. For further details see the Smart Contract section.

Context

According to the Hyperledger Fabric documentation the context contexts are used to “define and maintain user variables across transaction invocations within a smart contract”. It is a persistence / storage mechanism beyond the ledger which simplifies the logic of the program. Where transactions have a very limited lifespan the context will allow variables to be accessed after the transaction is completed. They help developers to create programs that are powerful, efficient and structured in such a way that they are easy to reason about. The context class also is defined within Hyperledger Fabric with a variety of methods which can be useful developers such as stub. Within the CommercialPaper example, and the UPMopoly game we have implemented a list of the objects is maintained which are the initiators or subjects of transactions. This helps us to reason about the subjects of the game – even if these are generally also stored in the ledger. Within the UPMopoly implementation a context is created in the GameContext.java file and the GameContext class. Here playerList and facultyList objects are created. These extend StateListImp which is an implementation of the StateList interface. These are part of the Hyperledger Fabric Ledger API package. The StateListImp class handles the actual interactions with the World State as well as the ledger (via the shim interface and stub object).

Smart Contract

At the heart of the game is the GameContract.java file. This is a smart contract which submits transactions to the ledger moving the players and faculties between states. The contract has several methods which implement the buy, sell, and pay rent transaction types as well as take care of other functions the players may need.

newPlayer adds a player to the game. It calls the createInstance method implemented in Player and passes the name, player number, starting amount and the initial state to the method. The player is added to the context class.

newFaculty adds a faculty to the game. Similar to createPlayer the createInstance method is called. The parameters which are passed include the faculty ID, name, the intended rental fee, the sale price and the initial state. The faculty is moved to the free state. Then it is added to the context before being returned to the caller.

buyFaculty changes the ownership of a faculty either in the initial purchase. The method accepts the context, player number and the faculty ID as parameters. With these it then retrieves the relevant faculty from the context by creating a key for the key-value store from the faculty ID. Before the faculty can be bought the availability is checked, an exception is thrown if this is not the case. The state is moved from free to bought and the owners player number is assigned to the faculty.

payRental is used to pay rent when a player lands on a faculty which is owned. In order to pay the rental fee, the function requires the context, faculty ID, player number of the owner and player number of the visitor (the player that has landed on the field. Based on the player numbers and faculty the method first constructs the keys in order to retrieve the player and faculty objects. It then finds the rental fee of the property. This amount is subtracted from the visitors and added to the owner. If the visitors balance is too low to pay their state is set to eliminated. The context is then updated.

facultySale is used for one player to sell a faculty to another. The method accepts the context, faculty ID, player number of the owner, player number of the buyer and the selling price as parameters. Again, the relevant players and faculty are retrieved from the context via their keys. The buyers balance is checked and an error is given if the funds are not available. The faculties owner is checked against the seller. The balances of each player are updated. Finally, the objects are passed back to the context in order to commit them to the ledger, world state and context lists.

printMoney shows the balance of a player. It takes the context and player number as inputs. A key is constructed from the player number and the player object is retrieved from the context. The players balance is then concatenated to a string indicating the balance.

printOwner shows the owner of a faculty or that it is unowned. It needs the context and faculty ID to be passed to it. From there it constructs a key with the faculty ID and retrieves the faculty object via the context. Via the Faculty getOwnerNumber method the owner's player ID is retrieved. From there a key for the player is constructed via this number. The player is retrieved and using the getName method the owner is determined and printed out.

printPlayer shows all the players who have not been eliminated from the game. The class proved extremely challenging to implement. Context is a class which is implemented in Hyperledger and the GameContext method inherits most of its accessible methods. The super class contains the getStub and getClientIdentity methods. ClientIdentity represents information about the identity which submitted a transaction. Neither appears immediately useful. The context class create a PlayerList and FacultyList object which are extensions of the StateList class which is a standard component of Hyperledger. StateList is however only an interface and has no usable methods. It is implemented in the StateListImp class. PlayerList and FacultyList both instantiate a StateListImp object. Unfortunetly StateListImp only contains methods for retrieving objects from the key-value store based on keys. Neither of us have

experience with key-value stores and it was not immediately apparent how to query the whole database. Adding to this complexity is the serialization and deserialization of objects before storing them.

The stub class is theoretically the solution. Stub offers an API to access the ledger and world state. Among the methods available via the class is the getPrivateDataByRange method which is send to "Returns all existing keys, and their values, that are lexicographically between startkey (inclusive) and the endKey (exclusive) in a given private collection." The method also allows one to pass empty strings as parameters which implies an unbounded range allowing us to retrieve all states. The method passes back a QueryResultsIterator<KeyValue> object. This object contains all state key and the corresponding data values. QueryResultsIterator is a customization of the iteration Java class. The KeyValue object is also a custom object. Because working with these is foreign to us, we decided to retrieve only the state key and use the existing classes in Player and PlayerList in order to query for the players which are still in the game. The statekeys are stored in an array and this is then iterated through. Players playing state is queried, if they are not eliminated the toString method is called and concatenated to a string which is finally returned. Unfortunately, in the implementation this did not work.

Instead, we opted to generate a string we know to correspond with the player number. The player numbers are always "no" with the integer appended in increments of one. We therefore use this logic to create a string with values "no1", "no2", "no3" and so on and so forth. Using this constructed key, we can access the players in the world state. We then use and if condition to check the player is not eliminated. If it passes this check the properties are called and appended to a string, which is finally returned.

Deployment

The deployment script brings the network live and publishes the contract to the ledger. In order to do so the bash script first compiles and installs the Java code using Gradle. To set up the network the network script is invoked (first with the down command in case the network is still up). A channel is created. Two peers are created as well as an orderer. Endorsements are given so that the peers can join the channel. The genesis block of the ledger is written. A role is assumed within the network in order to test the functionality.

The launch script executes the different methods of the smart contract in order to test these. Each method is called at least once, so the outputs can be seen in the console.