**Smart Contract Processing**

**Audience**: Architects, Application and smart contract developers

At the heart of a blockchain network is a smart contract. In PaperNet, the code in the commercial paper smart contract defines the valid states for commercial paper, and the transaction logic that transition a paper from one state to another. In this topic, we’re going to show you how to implement a real world smart contract that governs the process of issuing, buying and redeeming commercial paper.

We’re going to cover:

* [What is a smart contract and why it’s important](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/smartcontract.html#smart-contract)
* [How to define a smart contract](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/smartcontract.html#contract-class)
* [How to define a transaction](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/smartcontract.html#transaction-definition)
* [How to implement a transaction](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/smartcontract.html#transaction-logic)
* [How to represent a business object in a smart contract](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/smartcontract.html#representing-an-object)
* [How to store and retrieve an object in the ledger](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/smartcontract.html#access-the-ledger)

If you’d like, you can [download the sample](https://hyperledger-fabric.readthedocs.io/en/latest/install.html) and even [run it locally](https://hyperledger-fabric.readthedocs.io/en/latest/tutorial/commercial_paper.html). It is written in JavaScript and Java, but the logic is quite language independent, so you’ll easily be able to see what’s going on! (The sample will become available for Go as well.)

**Smart Contract**

A smart contract defines the different states of a business object and governs the processes that move the object between these different states. Smart contracts are important because they allow architects and smart contract developers to define the key business processes and data that are shared across the different organizations collaborating in a blockchain network.

In the PaperNet network, the smart contract is shared by the different network participants, such as MagnetoCorp and DigiBank. The same version of the smart contract must be used by all applications connected to the network so that they jointly implement the same shared business processes and data.

**Implementation Languages**

There are two runtimes that are supported, the Java Virtual Machine and Node.js. This gives the opportunity to use one of JavaScript, TypeScript, Java or any other language that can run on one of these supported runtimes.

In Java and TypeScript, annotations or decorators are used to provide information about the smart contract and it’s structure. This allows for a richer development experience — for example, author information or return types can be enforced. Within JavaScript, conventions must be followed, therefore, there are limitations around what can be determined automatically.

Examples here are given in both JavaScript and Java.

**Contract class**

A copy of the PaperNet commercial paper smart contract is contained in a single file. View it with your browser, or open it in your favorite editor if you’ve downloaded it.

* papercontract.js - [JavaScript version](https://github.com/hyperledger/fabric-samples/blob/master/commercial-paper/organization/magnetocorp/contract/lib/papercontract.js)
* CommercialPaperContract.java - [Java version](https://github.com/hyperledger/fabric-samples/blob/master/commercial-paper/organization/magnetocorp/contract-java/src/main/java/org/example/CommercialPaperContract.java)

You may notice from the file path that this is MagnetoCorp’s copy of the smart contract. MagnetoCorp and DigiBank must agree on the version of the smart contract that they are going to use. For now, it doesn’t matter which organization’s copy you use, they are all the same.

Spend a few moments looking at the overall structure of the smart contract; notice that it’s quite short! Towards the top of the file, you’ll see that there’s a definition for the commercial paper smart contract:

JavaScript

**class** CommercialPaperContract **extends** Contract {...}

Java

The CommercialPaperContract class contains the transaction definitions for commercial paper – **issue**, **buy** and **redeem**. It’s these transactions that bring commercial papers into existence and move them through their lifecycle. We’ll examine these [transactions](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/smartcontract.html#transaction-definition) soon, but for now notice for JavaScript, that the CommericalPaperContract extends the Hyperledger Fabric Contract [class](https://fabric-shim.github.io/release-1.4/fabric-contract-api.Contract.html).

With Java, the class must be decorated with the @Contract(...) annotation. This provides the opportunity to supply additional information about the contract, such as license and author. The @Default() annotation indicates that this contract class is the default contract class. Being able to mark a contract class as the default contract class is useful in some smart contracts which have multiple contract classes.

If you are using a TypeScript implementation, there are similar @Contract(...) annotations that fulfill the same purpose as in Java.

For more information on the available annotations, consult the available API documentation:

* [API documentation for Java smart contracts](https://hyperledger.github.io/fabric-chaincode-java/)
* [API documentation for Node.js smart contracts](https://fabric-shim.github.io/)

These classes, annotations, and the Context class, were brought into scope earlier:

JavaScript

**const** { Contract, Context } **=** require('fabric-contract-api');

Java

Our commercial paper contract will use built-in features of these classes, such as automatic method invocation, a [per-transaction context](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/transactioncontext.html), [transaction handlers](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/transactionhandler.html), and class-shared state.

Notice also how the JavaScript class constructor uses its [superclass](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super) to initialize itself with an explicit [contract name](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/contractname.html):

constructor() {

**super**('org.papernet.commercialpaper');

}

With the Java class, the constructor is blank as the explicit contract name can be specified in the @Contract() annotation. If it’s absent, then the name of the class is used.

Most importantly, org.papernet.commercialpaper is very descriptive – this smart contract is the agreed definition of commercial paper for all PaperNet organizations.

Usually there will only be one smart contract per file – contracts tend to have different lifecycles, which makes it sensible to separate them. However, in some cases, multiple smart contracts might provide syntactic help for applications, e.g. EuroBond, DollarBond, YenBond, but essentially provide the same function. In such cases, smart contracts and transactions can be disambiguated.

**Transaction definition**

Within the class, locate the **issue** method.

JavaScript

async issue(ctx, issuer, paperNumber, issueDateTime, maturityDateTime, faceValue) {...}

Java

The Java annotation @Transaction is used to mark this method as a transaction definition; TypeScript has an equivalent annotation.

This function is given control whenever this contract is called to issue a commercial paper. Recall how commercial paper 00001 was created with the following transaction:

Txn **=** issue

Issuer **=** MagnetoCorp

Paper **=** 00001

Issue time **=** 31 May 2020 09:00:00 EST

Maturity date **=** 30 November 2020

Face value **=** 5M USD

We’ve changed the variable names for programming style, but see how these properties map almost directly to the issue method variables.

The issue method is automatically given control by the contract whenever an application makes a request to issue a commercial paper. The transaction property values are made available to the method via the corresponding variables. See how an application submits a transaction using the Hyperledger Fabric SDK in the [application](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/application.html) topic, using a sample application program.

You might have noticed an extra variable in the **issue** definition – ctx. It’s called the [**transaction context**](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/transactioncontext.html), and it’s always first. By default, it maintains both per-contract and per-transaction information relevant to [transaction logic](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/smartcontract.html#transaction-logic). For example, it would contain MagnetoCorp’s specified transaction identifier, a MagnetoCorp issuing user’s digital certificate, as well as access to the ledger API.

See how the smart contract extends the default transaction context by implementing its own createContext() method rather than accepting the default implementation:

JavaScript

createContext() {

**return** **new** CommercialPaperContext()

}

Java

This extended context adds a custom property paperList to the defaults:

JavaScript

**class** CommercialPaperContext **extends** Context {

constructor() {

**super**();

*// All papers are held in a list of papers*

**this**.paperList **=** **new** PaperList(**this**);

}

Java

We’ll soon see how ctx.paperList can be subsequently used to help store and retrieve all PaperNet commercial papers.

To solidify your understanding of the structure of a smart contract transaction, locate the **buy** and **redeem** transaction definitions, and see if you can see how they map to their corresponding commercial paper transactions.

The **buy** transaction:

Txn **=** buy

Issuer **=** MagnetoCorp

Paper **=** 00001

Current owner **=** MagnetoCorp

New owner **=** DigiBank

Purchase time **=** 31 May 2020 10:00:00 EST

Price **=** 4.94M USD

JavaScript

async buy(ctx, issuer, paperNumber, currentOwner, newOwner, price, purchaseTime) {...}

Java

The **redeem** transaction:

Txn **=** redeem

Issuer **=** MagnetoCorp

Paper **=** 00001

Redeemer **=** DigiBank

Redeem time **=** 31 Dec 2020 12:00:00 EST

JavaScript

async redeem(ctx, issuer, paperNumber, redeemingOwner, redeemDateTime) {...}

Java

In both cases, observe the 1:1 correspondence between the commercial paper transaction and the smart contract method definition.

All of the JavaScript functions use the async and await [keywords](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) which allow JavaScript functions to be treated as if they were synchronous function calls.

**Transaction logic**

Now that you’ve seen how contracts are structured and transactions are defined, let’s focus on the logic within the smart contract.

Recall the first **issue** transaction:

Txn **=** issue

Issuer **=** MagnetoCorp

Paper **=** 00001

Issue time **=** 31 May 2020 09:00:00 EST

Maturity date **=** 30 November 2020

Face value **=** 5M USD

It results in the **issue** method being passed control:

JavaScript

async issue(ctx, issuer, paperNumber, issueDateTime, maturityDateTime, faceValue) {

*// create an instance of the paper*

**let** paper **=** CommercialPaper.createInstance(issuer, paperNumber, issueDateTime, maturityDateTime, faceValue);

*// Smart contract, rather than paper, moves paper into ISSUED state*

paper.setIssued();

*// Newly issued paper is owned by the issuer*

paper.setOwner(issuer);

*// Add the paper to the list of all similar commercial papers in the ledger world state*

await ctx.paperList.addPaper(paper);

*// Must return a serialized paper to caller of smart contract*

**return** paper.toBuffer();

}

Java

The logic is simple: take the transaction input variables, create a new commercial paper paper, add it to the list of all commercial papers using paperList, and return the new commercial paper (serialized as a buffer) as the transaction response.

See how paperList is retrieved from the transaction context to provide access to the list of commercial papers. issue(), buy() and redeem() continually re-access ctx.paperList to keep the list of commercial papers up-to-date.

The logic for the **buy** transaction is a little more elaborate:

JavaScript

async buy(ctx, issuer, paperNumber, currentOwner, newOwner, price, purchaseDateTime) {

*// Retrieve the current paper using key fields provided*

**let** paperKey **=** CommercialPaper.makeKey([issuer, paperNumber]);

**let** paper **=** await ctx.paperList.getPaper(paperKey);

*// Validate current owner*

**if** (paper.getOwner() **!==** currentOwner) {

**throw** **new** Error('Paper ' **+** issuer **+** paperNumber **+** ' is not owned by ' **+** currentOwner);

}

*// First buy moves state from ISSUED to TRADING*

**if** (paper.isIssued()) {

paper.setTrading();

}

*// Check paper is not already REDEEMED*

**if** (paper.isTrading()) {

paper.setOwner(newOwner);

} **else** {

**throw** **new** Error('Paper ' **+** issuer **+** paperNumber **+** ' is not trading. Current state = ' **+**paper.getCurrentState());

}

*// Update the paper*

await ctx.paperList.updatePaper(paper);

**return** paper.toBuffer();

}

Java

See how the transaction checks currentOwner and that paper is TRADING before changing the owner with paper.setOwner(newOwner). The basic flow is simple though – check some pre-conditions, set the new owner, update the commercial paper on the ledger, and return the updated commercial paper (serialized as a buffer) as the transaction response.

Why don’t you see if you can understand the logic for the **redeem** transaction?

**Representing an object**

We’ve seen how to define and implement the **issue**, **buy** and **redeem** transactions using the CommercialPaper and PaperList classes. Let’s end this topic by seeing how these classes work.

Locate the CommercialPaper class:

JavaScriptIn the [paper.js file](https://github.com/hyperledger/fabric-samples/blob/master/commercial-paper/organization/magnetocorp/contract/lib/paper.js):

**class** CommercialPaper **extends** State {...}

Java

This class contains the in-memory representation of a commercial paper state. See how the createInstance method initializes a new commercial paper with the provided parameters:

JavaScript

**static** createInstance(issuer, paperNumber, issueDateTime, maturityDateTime, faceValue) {

**return** **new** CommercialPaper({ issuer, paperNumber, issueDateTime, maturityDateTime, faceValue });

}

Java

Recall how this class was used by the **issue** transaction:

JavaScript

**let** paper **=** CommercialPaper.createInstance(issuer, paperNumber, issueDateTime, maturityDateTime, faceValue);

Java

See how every time the issue transaction is called, a new in-memory instance of a commercial paper is created containing the transaction data.

A few important points to note:

* This is an in-memory representation; we’ll see [later](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/smartcontract.html#accessing-the-ledger) how it appears on the ledger.
* The CommercialPaper class extends the State class. State is an application-defined class which creates a common abstraction for a state. All states have a business object class which they represent, a composite key, can be serialized and de-serialized, and so on. State helps our code be more legible when we are storing more than one business object type on the ledger. Examine the State class in the state.js [file](https://github.com/hyperledger/fabric-samples/blob/master/commercial-paper/organization/magnetocorp/contract/ledger-api/state.js).
* A paper computes its own key when it is created – this key will be used when the ledger is accessed. The key is formed from a combination of issuer and paperNumber.
* constructor(obj) {
* **super**(CommercialPaper.getClass(), [obj.issuer, obj.paperNumber]);
* Object.assign(**this**, obj);
* }
* A paper is moved to the ISSUED state by the transaction, not by the paper class. That’s because it’s the smart contract that governs the lifecycle state of the paper. For example, an import transaction might create a new set of papers immediately in the TRADING state.

The rest of the CommercialPaper class contains simple helper methods:

getOwner() {

**return** **this**.owner;

}

Recall how methods like this were used by the smart contract to move the commercial paper through its lifecycle. For example, in the **redeem** transaction we saw:

**if** (paper.getOwner() **===** redeemingOwner) {

paper.setOwner(paper.getIssuer());

paper.setRedeemed();

}

**Access the ledger**

Now locate the PaperList class in the paperlist.js [file](https://github.com/hyperledger/fabric-samples/blob/master/commercial-paper/organization/magnetocorp/contract/lib/paperlist.js):

**class** PaperList **extends** StateList {

This utility class is used to manage all PaperNet commercial papers in Hyperledger Fabric state database. The PaperList data structures are described in more detail in the [architecture topic](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/architecture.html).

Like the CommercialPaper class, this class extends an application-defined StateList class which creates a common abstraction for a list of states – in this case, all the commercial papers in PaperNet.

The addPaper() method is a simple veneer over the StateList.addState() method:

async addPaper(paper) {

**return** **this**.addState(paper);

}

You can see in the StateList.js [file](https://github.com/hyperledger/fabric-samples/blob/master/commercial-paper/organization/magnetocorp/contract/ledger-api/statelist.js) how the StateList class uses the Fabric API putState() to write the commercial paper as state data in the ledger:

async addState(state) {

**let** key **=** **this**.ctx.stub.createCompositeKey(**this**.name, state.getSplitKey());

**let** data **=** State.serialize(state);

await **this**.ctx.stub.putState(key, data);

}

Every piece of state data in a ledger requires these two fundamental elements:

* **Key**: key is formed with createCompositeKey() using a fixed name and the key of state. The name was assigned when the PaperList object was constructed, and state.getSplitKey() determines each state’s unique key.
* **Data**: data is simply the serialized form of the commercial paper state, created using the State.serialize() utility method. The State class serializes and deserializes data using JSON, and the State’s business object class as required, in our case CommercialPaper, again set when the PaperList object was constructed.

Notice how a StateList doesn’t store anything about an individual state or the total list of states – it delegates all of that to the Fabric state database. This is an important design pattern – it reduces the opportunity for [ledger MVCC collisions](https://hyperledger-fabric.readthedocs.io/en/latest/readwrite.html) in Hyperledger Fabric.

The StateList getState() and updateState() methods work in similar ways:

async getState(key) {

**let** ledgerKey **=** **this**.ctx.stub.createCompositeKey(**this**.name, State.splitKey(key));

**let** data **=** await **this**.ctx.stub.getState(ledgerKey);

**let** state **=** State.deserialize(data, **this**.supportedClasses);

**return** state;

}

async updateState(state) {

**let** key **=** **this**.ctx.stub.createCompositeKey(**this**.name, state.getSplitKey());

**let** data **=** State.serialize(state);

await **this**.ctx.stub.putState(key, data);

}

See how they use the Fabric APIs putState(), getState() and createCompositeKey() to access the ledger. We’ll expand this smart contract later to list all commercial papers in paperNet – what might the method look like to implement this ledger retrieval?

That’s it! In this topic you’ve understood how to implement the smart contract for PaperNet. You can move to the next sub topic to see how an application calls the smart contract using the Fabric SDK.