Make your blockchain smart contracts smarter with business rules

Stephane Mery
Daniel Selman

August 14, 2017

This article describes how to implement blockchain smart contracts, using business rules on top of Hyperledger Fabric. Work with sample code to learn how to model and execute smart contract decision logic, taking advantage of an enterprise-class rule engine to improve transparency and trust for all participants in a business network.

Blockchain already has a profound impact on multi-party business processes. Organizations that use blockchain count on trusted automatic transactions. It provides a framework of trust that you can use to securely automate processes that, until now, were often manual.

Traditionally, in business transactions, when two parties exchange value, they need to share a representation of the exchanged value and of the terms and conditions for the transaction. If they can't fully trust each other, each party maintains its own record of the exchange – a transaction ledger. They also keep their own copies of the rules and processes that govern the contract to exchange value.

Duplicating these records can lead to errors – and fraud. On top of this, the transaction processes are also duplicated, and performed manually and inefficiently. Disputes over discrepancies between the copies of the ledgers and contracts might need to be resolved in court, leading to significant costs, and ultimately delays to the transaction.

Blockchain uses cryptography along with distributed computing and storage technologies to solve the problems inherent in the traditional processes by providing trading parties with a means to share a trusted representation of their transactions and their assets.

Value exchanges follow rules that the parties agree on beforehand through a legally binding contract such as in a sell/buy transaction where the parties exchange the ownership of an asset for an agreed amount of money. However, often the rules are more complex and include laws and regulations from authorities that have jurisdiction over the transaction.

With blockchain technology, the parties use smart contracts to encode and encapsulate the transaction rules and processes that govern transactions. When the transaction is processed, the
smart contract automatically performs actions and compliance checks according to the rules they have defined.

Integrate a decision management platform

You can make your smart contracts even smarter if you implement them by using a rule engine supported by a decision management platform. With IBM® Operational Decision Manager (ODM), you can add trust at the smart contract definition level. This approach enables business stakeholders to contribute in the implementation of the smart contract decision logic. It fully complies to the technical trust model of the blockchain with its insertion into the distributed transaction system of Hyperledger® Fabric™.

This article describes the integration of IBM ODM with Hyperledger Composer, an open source blockchain development framework, running on Hyperledger Fabric, a blockchain platform. Hyperledger Composer and Fabric are two of the Hyperledger projects hosted by The Linux Foundation. Learn how to use IBM ODM in blockchain solutions, to bring value to all participants in a distrusted business network.

The Vehicle Lifecycle sample

To illustrate how IBM ODM can bring value to a blockchain network, the following sample considers the lifecycle of a vehicle. The illustration follows vehicles from manufacture to recycling, through registration with the vehicle regulatory authority, and transfer of ownership when the vehicle is sold.

This sample is based on the one included with Hyperledger Composer. The sample integrates IBM ODM business decision logic (rules) in smart contracts that run on Hyperledger Fabric. Both the Hyperledger Composer sample, and sample integrating IBM ODM are available on GitHub:


Multiple parties are involved in the lifecycle of a vehicle, including the manufacturer, car dealers, vehicle regulatory authority, insurance companies, owners, and scrap dealers. All parties in the business network benefit from having a trusted, distributed, source of truth about the history and ownership of a vehicle.
The vehicle itself needs to be described with the characteristics that are important for each participant, such as its technical and commercial characteristics, owner identification, and insurance status.

Transactions related to a vehicle include the initial purchase order, manufacturing order, request for registration, and ownership transfer. These transactions are governed by different rules at different stages of the vehicle lifecycle. Business rules likely apply to the sales process when the car is initially purchased from the car dealer. Change of ownership likely needs to be compliant to trade laws related to the vehicle, specific to each country. Participants want to detect fraudulent transactions on non-compliant vehicles or other tax evasion patterns.

Some of the transaction processing, embedded in smart contracts, is largely technical in nature, adapting and transforming data structures, as well as creating, updating, and destroying assets. Other parts of the smart contract logic are much closer to the legal rules that govern the underlying contract between the parties or by the government regulator.

The code in this example requires IBM ODM V8.9.x and Hyperledger Composer V0.11.1 or more recent versions. For all other requirements, see the README file for the sample code.

The sample code includes the projects shown in the following illustration, providing both the sample application and the necessary infrastructure code that enable IBM ODM runtime components in a blockchain infrastructure.

To deploy and run the sample, see the README file in GitHub.

**Smart contracts**

Smart contracts are fundamental to blockchain platforms. With smart contracts, you can securely apply rules when you process transactions. You can use them to automatically perform validation steps and encode conditions that in the past were on a signed physical contract.

To establish a smart contract, participants in a blockchain platform agree on how transactions and their data are represented, and on the rules that govern these transactions. This agreement involves expressing these rules precisely, exploring all possible exceptions, and defining a
framework for resolving disputes. It is usually an iterative process involving developers and business stakeholders.

The process also covers reviewing the rules, registering agreement between the parties, testing the rules on transaction data, simulating scenarios to understand their business impact, and storing them in a secure and transparent way. In addition, the same attention must be applied to data models and the business domain models they represent. The parties also must define how rules are governed: who can define rules, who can deploy rules, and the processes for changing rules.

Smart contracts manipulate value, so it is critical that you use the right tools and practices to develop safe, secured, transparent and logic that can be trusted – all automated inside the platform to speed up the transaction and realize the promise of automation.

The processes governing smart contracts are like those found in decision management platforms such as IBM ODM, where operational decisions are represented, automated and governed as business assets, and handled by business stakeholders.

This article explores how you can represent smart contract business rules in IBM ODM, manage them through the extensive IBM ODM tool set, and run them with a rule engine inside the blockchain platform.

**Topology of a blockchain network with Hyperledger Fabric, Hyperledger Composer, and IBM ODM**

This section explores how to insert a rule engine into the blockchain network architecture to add rule execution capabilities to smart contracts.

For information on the general architecture of a blockchain network, see the [Hyperledger Fabric documentation](https://hyperledger-fabric.readthedocs.io/) and the [Hyperledger Composer documentation](https://hyperledger-composer.readthedocs.io/).

The following figure shows the typical architecture of a blockchain business network. Each peer node has its own instance of the IBM ODM Rule Execution Server, which runs rules for all blockchain applications deployed on it.
A blockchain network runs on a set of nodes. Each node has a copy of the transaction ledger and assets that is stored in a database, called *world state* in Hyperledger Fabric terminology.

Inside a node, several processes implement the blockchain protocols and functions. A peer node in this topology is the set of processes required to handle the transactions on a blockchain node.

Each Hyperledger Composer application in the business network is represented by a chaincode process. The chaincode has a JavaScript interpreter that executes the logic for processing the transaction. Chaincodes are the mechanism through which smart contracts are defined in the HyperLedger Fabric blockchain implementation.

Peer node processes can be physically run on different machines to ensure high availability of a given blockchain node. If one of the peer nodes fails, the second one can still validate the incoming transactions.

Business network nodes must be protected by strong isolated security schemes, so you need to deploy an instance of IBM ODM Rule Execution Server for each peer node. Multiple peer nodes make both the application and the rule execution capability highly available.

Deploy blockchain nodes with Docker Compose, using Docker images of Hyperledger Fabric and Hyperledger Composer.

The following diagram shows the typical architecture of a peer node that serves several blockchain applications:
Deploying Hyperledger Fabric and Hyperledger Composer

As a blockchain application development framework, Hyperledger Composer provides the necessary scripts to set up a Hyperledger Fabric and Hyperledger Composer blockchain infrastructure. The infrastructure is deployed as a set of Docker containers, which gives you a lot of flexibility for how you deploy a business network infrastructure.

For information on how to set up a running blockchain infrastructure on your own machine or a set of machines, see the Hyperledger Composer documentation.

Deploying the IBM ODM Rule Execution Server

The samples in this article complement the Hyperledger Fabric and Hyperledger Composer infrastructure with two new components: an IBM ODM Rule Execution Server and a Deployer process. The Rule Execution Server executes decisions (business rules) in the smart contracts and the Deployer supports the deployment of the business rules through the blockchain.

The Deployer interacts with the Rule Execution Server management API. You can think of it as a dedicated facade for the Rule Execution Server that provides the APIs that smart contracts invoke.

To run the Rule Execution Server in the Docker-based Hyperledger Fabric infrastructure, package it as a Docker image. See the Deploy an IBM Operational Decision Manager topology with Docker Compose developerWorks article.

The `odm-runtime` component of the Vehicle Lifecycle sample contains the Docker file used to build an IBM ODM Docker image and instructions on how to deploy the Rule Execution Server to a Hyperledger Fabric and Hyperledger Composer default infrastructure.

The `odm-deployer` component of the Vehicle Lifecycle sample contains a Node.js implementation of this deployment facade. Smart contracts use it to deploy the Java executable object model (XOM) and the rules used by the decision services that are referenced by the smart contracts.
After you deploy and run the Rule Execution Server Docker image and the Deployer Docker images in the blockchain business network, it is possible to implement rule-based smart contracts that take advantage of IBM ODM capabilities.

**The Vehicle Lifecycle data model in Hyperledger Composer**

Hyperledger Composer has several concepts for representing data and defining the transaction processing logic:

- **Participants** represent actors that interact with the blockchain application through transactions.
- **Assets** model the items to be stored in the blockchain. They often represent something of value manipulated by the transactions.
- **Transactions** represent the actual transaction registered on the blockchain ledger, initiated from a participant, related to one or more assets.
- **Transaction processors** are used when a transaction is initiated on the blockchain and all nodes of the blockchain validate it and perform side effects. In Hyperledger Composer, transaction processor callbacks are called in all nodes of the blockchain. Consensus algorithms check that all nodes process the transaction the same way, which ensures a consistent update of the ledger and the world state database.

Participants, assets, and transactions are described in Hyperledger Composer by using a dedicated high level language. Transaction processors are implemented as JavaScript functions that manipulate the resources.

For more information on Hyperledger Composer applications, see [Business Network Definition](#) in the Hyperledger Composer documentation.

The Hyperledger Composer Playground web user interface allows you to browse the data model of the different resources, as well as the JavaScript code of the transaction processors.

**Participant example**

The following code describes the model of a `Person`, which in this example is someone who can buy or sell a `vehicle`.
You can create instances of Person in the blockchain databases by using the REST API or through specific transactions. They are stored as JavaScript Object Notation (JSON) documents. For example, the following document describes a Person called Dan:

![Person JSON document](https://ibm.com/developerWorks)

**Asset example**

The main asset manipulated by this sample is a Vehicle, shown in the following screen capture. The object model describing a vehicle is quite complex, because it needs to account for all the properties used in the diverse range of transactions that involve a vehicle.

![Vehicle object model](https://ibm.com/developerWorks)

An instance of Vehicle is represented as a JSON document. It is depicted in the following screen capture in Hyperledger Composer Playground, which shows the Vehicle registry:

![Vehicle registry](https://ibm.com/developerWorks)

**Transaction example**

The transfer of ownership of a Vehicle from one Person to another is performed as a transaction registered on the blockchain.

![Transaction example](https://ibm.com/developerWorks)
The `PrivateTransferTransaction` class, shown in the following screen capture, represents this transaction in Hyperledger Composer:

```java
abstract class Transaction : TransactionModel {
    constructor
    |
    + vehicle
    |
    +...}

class VehicleTransaction {
    + vehicle seller
    + vehicle buyer
    + String validates optional
}
```

An example of this transaction is represented by the following JSON document:

```json
{
    "schema": "org.w3c.ProtectedPrivateVehicleTransaction",
    "vehicle": "resources.composers.weak.PrivateVehicle",
    "vehicle Seller": "resources.composers.weak.Proprietor",
    "vehicle Buyer": "resources.composers.weak.Proprietor",
    "vehicle": "resources.composers.weak.Vehicle16497095a"
}
```

You can find the data model for this example in the `model` folder of the `vehicle-lifecycle sample`, described in the different `.cto` files.

**The Vehicle Lifecycle data model in IBM ODM**

You also need to model the data so it can be used by business rules in IBM ODM. There are three layers of data represented in IBM ODM:

- The Java executable object model (XOM), which is implemented as a set of Java classes in this example. This model represents the data that are used inside the rule engine to apply the decision logic.
- The business object model (BOM), which represents the data model at a more abstract level. Typically, the BOM reflects the XOM, but you can define more business-friendly concepts and methods to support a natural language that business stakeholders use to specify their decision logic.
- The business vocabulary. This language-specific layer allows you to define the business terms and the phrases that refer to data in business rules.

You define the different layers in IBM ODM Rule Designer. See the IBM ODM documentation for more information. [Designing business object models](#) describes the XOM, BOM, and vocabulary concepts.

For example, the `Vehicle` is represented by the following Java class in the XOM:

```java
package org.w3c.
import org.w3c.composers.weak.private.weak.
import com.composers.weak.PrivateVehicle
import public.composers.weak.Vehicle

abstract class VehicleTransaction {
    + vehicle seller
    + vehicle buyer
    + String validates optional
}
```

The `Vehicle` is represented by the following BOM class, along with its English verbalization:
Typically, the data model is created in Hyperledger Composer and a Java developer creates a Java-based implementation of it that is used as the XOM. The Java developer works with a business analyst in Rule Designer to create the BOM and vocabulary, starting from the Java-based XOM.

Note that Hyperledger Composer provides a tool to generate a Java representation from the cto files of the domain model (which could be a good starting point for building an XOM).

For more information about this capability, see the README file in the vehicle-lifecycle-xom directory of the sample.

IBM ODM V8.9.x supports Jackson to serialize and deserialize the Java object model from JSON. JSON is used as a pivot data exchange format between the HyperLedger Composer smart contract and the IBM ODM decision services.

To ensure that Hyperledger Composer transaction processors and IBM ODM decision services can easily exchange data, it’s essential that the data model implemented in Hyperledger Composer matches the XOM used in the decision service.

See the Java implementation in vehicle-lifecycle-xom to understand how the XOM should be implemented to allow a smooth exchange of data between Hyperledger Composer and IBM ODM.

**The lifecycle of an ownership transfer transaction**

To illustrate how the blockchain application and IBM ODM work together, consider the case of a sell/buy transaction between two PersonS.

To perform this business transaction, a system needs to publish a VehicleTransferTransaction to the blockchain with all the information about the Vehicle, the seller, and the buyer.

Assume that the transaction is governed by rules that check if the transaction is fraudulent. The logic for these rules is complex and defined by experts. When new fraud patterns are discovered, the experts (business stakeholders) want to quickly change the rules to automatically identify fraudulent transactions as soon as they are submitted to the blockchain.

The transaction goes through the journey shown in the following illustration:
• An external system, which could be a business process or an application, uses the REST API generated by Hyperledger Composer for this blockchain application to push a `PrivateVehicleTransfer` transaction.
• The transaction is distributed by Hyperledger Fabric to all nodes in the blockchain.
• Each peer node triggers the chaincode (smart contract) for this blockchain application to check the transaction and apply the logic implemented by the smart contract.
• The chaincode triggers the Hyperledger Composer transaction processor callback that is defined in the blockchain application.
• The transaction processor JavaScript code makes an external REST call to the Rule Execution Server that holds the business rules in a decision service, passing the data necessary to take the decision.
• The business rules are applied to the transaction and executed by the rule engine.
• The response is sent back to the Hyperledger Composer callback.
• The callback performs actions according to the response, either rejecting the transaction or validating it, with extra information in the case of a suspicious transaction.

Note that this article focuses on the interaction between the Hyperledger Composer JavaScript callback, which delegates to IBM ODM all the business-centric logic. For information on how to submit transactions to Hyperledger Fabric or how they are handled by blockchain, see to the Hyperledger Composer and Hyperledger Fabric documentation.

**Call the decision service**

The Hyperledger Composer chaincode is not running in the same process as the IBM ODM rule engine, so the two processes communicate through REST calls.
Hyperledger Composer provides an API that you use to perform a POST REST call outside the chaincode, passing the current transaction, along with any other assets or participant data that is required. Hyperledger Composer transforms the resources passed as parameters to the REST call into a JSON document.

IBM ODM exposes a decision service as a REST API that can take a JSON payload describing the request payload.

For example, the following Hyperledger Composer transaction processor uses a decision service to determine if the given transaction is fraudulent or not, as shown in the following code:

```javascript
/**
 * Transfer a vehicle to another private owner
 * @param {org.vda.PrivateVehicleTransfer} privateVehicleTransfer - the PrivateVehicleTransfer transaction
 * @transaction
 */
function privateVehicleTransfer(privateVehicleTransfer) {
    ...
}
```

The JavaScript code knows which decision service implements the decision it needs to exercise. As part of the peer node execution environment, a Rule Execution Server is running, ready to receive requests through the REST API.

Use the following code in the callback to invoke the decision service:

```javascript
var rulesetPath = ruleappName + "" + currentRuleappVersion + "" + rulesetName + "" + currentRulesetVersion;
    // this is where we're calling out to a Decision Service to determine if the transaction is suspicious or not
    // The Decision Service returns a 'status' and a 'message'. 'status' could be ACCEPTED, REJECTED, SUSPICION.
    // If REJECTED, the transaction should abort with the 'message' indicating why. If SUSPICION, the 'message'
    // should be assigned to the Vehicle.suspiciousMessage field
    // The Decision Service receives all the data about the current transaction: buyer, seller and the vehicle

    var url = 'http://odmruntime_odm-runtime_1:9060/DecisionService/rest/' + rulesetPath;
    var dsCallObject = factory.newResource(NS_DECISION, 'IsSuspiciousTransferDecisionService',
    "isSuspiciousTransfer");
    dsCallObject.transaction = privateVehicleTransfer;

    print("Calling ODM Decision Service: " + url);
    return post( url, dsCallObject, {permitResourcesForRelationships: true, deduplicateResources: true});
```

You can see the code for this example in vda.js.

The transaction processed by this callback is wrapped in a dedicated object and passed to the call. The wrapper object is introduced so that the JSON payload sent by Hyperledger Composer matches the request payload expected by the decision service, which depends on the decision service signature specified in IBM ODM. For example, a decision service signature can have several input parameters that all must be entries in the request payload.
The transaction entry in the following payload matches the transaction input parameter specified in the decision operation in IBM ODM.

As the `PrivateVehicleTransfer` transaction points to the buyer, the seller and the vehicle, the entire object graph is serialized as a JSON document. It is sent to the decision service when calling the Hyperledger Composer `post()` API. For this transaction, the following JSON code is generated:

```json
{
    "$class": "org.acme.vehicle.lifecycle.decision.IsSuspiciousTransferDecisionService",
    "$id": 
        "resource:org.acme.vehicle.lifecycle.decision.IsSuspiciousTransferDecisionService#isSuspiciousTransfer",
    "transaction": {
        "$class": "org.vda.PrivateVehicleTransfer",
        "$id": 
            "resource:org.vda.PrivateVehicleTransfer#4302c409-96f1-4660-9772-400875e5e2e2",
        "seller": {
            "$class": "composer.base.Person",
            "$id": "resource:composer.base.Person#dan",
            "ssn": "dan",
            "firstName": "Dan",
            "lastName": "Selman",
            "gender": "MALE",
            "nationalities": [
                "French",
                "UK"
            ],
            "contactDetails": {
                "$class": "composer.base.ContactDetails",
                "email": "dan@acme.org",
                "address": {
                    "$class": "composer.base.Address",
                    "city": "Winchester",
                    "country": "UK",
                    "region": "England"
                }
            }
        },
        "buyer": {
            "$class": "composer.base.Person",
            "$id": "resource:composer.base.Person#anthony",
            "ssn": "anthony",
            "firstName": "Anthony",
            "lastName": "Smith",
            "gender": "MALE",
            "nationalities": [
                "French"
            ],
            "contactDetails": {
                "$class": "composer.base.ContactDetails",
                "email": "anthony@acme.org",
                "address": {
                    "$class": "composer.base.Address",
                    "city": "Paris",
                    "country": "France",
                    "region": "IDF"
                }
            }
        },
        "specialNotes": "Dan selling a car to Anthony Smith",
        "vehicle": {
            "$class": "org.vda.Vehicle",
            "$id": "resource:org.vda.Vehicle#156478954",
            "vin": "156478954",
            ...
        }
    }
}
```
The XOM for the decision service should support this payload. Hyperledger Composer serializes objects on the first level and then refers to objects which have already been serialized through IDs. Specific JSON annotations in the XOM allow to handle object graph deserialization. See the code for the vehicle-lifecycle-xom project.

Make sure you call the right version of the decision service, because the decision logic might evolve independently of the code in the smart contract. The same version of the decision logic must be used by all nodes of the blockchain network. A way to manage versioning the decision logic is described in the lifecycle illustration.

The decision service returns a JSON document that describes the decision action. If the format of the input JSON document is driven by Hyperledger Composer and the form of the data model of the Blockchain application, the format of the answer is free and controlled by the IBM ODM decision service. In the examples in this article, the decision service returns a JSON payload like the following code, if there is a suspicious transaction:

```json
{
  "status": "SUSPICION",
  "message": "Cross Border Suspicious Transfer: please double check buyer regulation"
}
```

This response is used by the transaction processor to carry on the processing of the transaction according to the business decision that has been taken by the decision service. In the following example, the result of the decision is written into the vehicle asset.
Implement the decision logic

The decision logic is implemented as a regular IBM ODM decision service. You can define several entry points to a decision service to accommodate the different business decisions that you need to delegate to IBM ODM in your smart contracts. An entry point, which corresponds to a decision operation in IBM ODM, is a dedicated ruleset that has a signature and contains the decision logic expressed as business rules.

The input parameter of the decision is usually a transaction from Hyperledger Composer, referring to assets, participants, or historical transactions that you need to complete for the decision. If you need to pass more data, you need to model a dedicated resource object in Hyperledger Composer (an asset or a concept) and populate it in the Transaction Processor before invoking the decision service.

The output parameter is an object that holds the decision and any data that you need to return to the smart contract.

In the samples for this article, the entry point is defined by the decision operation in following screen capture:

The REST API created from this decision operation supports the JSON input serialized by Hyperledger Composer when calling the `post` API.

The output parameter is a data structure with a `status` and a `message` that is populated by the business rules and returned as a JSON document to the caller.

The decision logic in the sample is straightforward: a ruleflow, with a business rule, and a decision table, shown in the following three screen captures.

The ruleflow in IBM ODM Rule Designer:
IBM ODM allows for decision logic as complex as you require. Decision logic made of thousands or tens of thousands of rules is common and easily supported by IBM ODM. The sample in this article mostly uses Rule Designer to author the decision logic, but you can also use Decision Center to define and manage the business logic. Look for future content about leveraging Decision Composer to implement the decision logic and collaboratively manage the lifecycle of your business rules.

Note that the business vocabulary used in the rules can be fully specified in the IBM ODM BOM editor. For example, in this sample, the data model given by Hyperledger Composer is quite verbose with several levels of nesting. The BOM is adapted to implement shortcut phrases to hide this complexity. In the sample, the `Person` concept has an extra `getCountry()` method that hides the underlying complexity of finding the country information, shown in the following screen capture:
Decision logic lifecycle

Blockchain is a runtime environment where the processing of transactions through smart contracts is, by design, distributed and controlled by mechanisms that ensure the consistency and the security of the ledger.

When a transaction is submitted to Hyperledger Fabric, it is dispatched to all nodes of the blockchain networks and several instances of the smart contracts apply to process the transactions. All instances of these smart contracts must return the same set of updates of Assets they controlled, and they must take the same decision about the validity of the transaction. So it is important that all nodes have the same version of the decision logic delegated to IBM ODM. Also, just like the smart contract code is tamper-proof inside the blockchain, it is important that no-one can tamper with the business rules.

For these two reasons, deploying a new version of the decision logic cannot be handled through the typical deployment process usually used with IBM ODM. Because all nodes have their own hidden and private instances of the Rule Execution Server, deployment from the Rule Execution Server Console, Rule Designer, or Decision Center (or through any other centrally-managed APIs in IBM ODM API) is not applicable to blockchain use cases.

The vehicle-lifecycle sample features a dedicated deployment mechanism, which uses features of blockchain to ensure that all nodes have the same version of the logic.

Specific assets and transactions are added to the application to store and manage the rulesets. You can see their implementation in odm.ctl and odm.js in the vehicle-lifecycle project.

Updates to the decision logic are handled as a blockchain transaction. When the decision logic changes, IBM ODM generates a new version of the ruleset archive. The binary of this archive, as well as its version number, are submitted to the blockchain through a transaction, as shown in the following code:

```plaintext
transaction RuleAppUpdated identified by transactionId
{
  o String transactionId
  o String resDeployerURL
  // ruleapp name should be of the form <ruleapp>/<ruleset>
  o String ruleAppName
  o String ruleapp_version
  o String ruleset_version
  o String archive
  o String managedXomURI
}
```

A dedicated Transaction Processor processes the transaction on each blockchain node. It performs the following three actions:

- It stores the archive in the world state database that is controlled by the blockchain:
• It writes the current version of the RuleApp or the ruleset to use for a given decision service in a specific asset (RuleAppCurrentVersion):

```java
asset RuleAppCurrentVersion identified by ruleAppName
{
    // ruleapp name should be of the form <ruleapp>/<ruleset>
    o String ruleAppName
    o String ruleapp_version
    o String ruleset_version
}
```

• It deploys the RuleApp or the ruleset to the Rule Execution Server for the node through the Deployer facade, passing the Archive binary along with version information.

To know which version of the ruleset to use, a Transaction Processor looks up this information from this specific asset and uses it to parameterize the REST API when it invokes the decision service.

If something goes wrong and the RuleAppUpdated transaction needs to be rolled-back, the Current Version asset is also rolled-back, and the previous version of the decision service is still used. Then, the newest version of the ruleset is deployed to the Rule Execution Server but it is never used.

This mechanism can adapted to avoid storing large binary RuleApps in the ledger directly. The ledger can store a reference to the Ruleapp with a hashcode signature that the smart contract can use to check that the binary is not modified.

Note that the decision logic highly depends on the XOM supporting the business rules. A similar mechanism is illustrated in the sample to deploy the XOM to the various Rule Execution Servers that run in the blockchain.

**Decision logic management and governance**

One key advantage of using IBM ODM for smart contracts is the extensive tools that business stakeholders can use to manage and govern the decision logic of the smart contracts. The decision logic usually evolves faster than the rest of the application according to weekly or even daily changes in the market, or other factors.

It is important that representatives of the parties involved can control the logic in the smart contract. If the smart contract implements a concrete contract that regulates how the parties exchange assets and values, they must agree on the rules and on the lifecycle.
For example, the parties might decide to review the rules together, negotiate the content, and agree when these rules should be deployed. This agreement likely requires a well-defined governance process, with several levels of review and approval from each party.

IBM ODM excels in this area with the Decision Center and its business console, shown in the following screen capture:

![Decision Center screenshot]

Even though rule execution is necessarily distributed for a blockchain platform, you can centralize the management environment, which makes it easier for the business stakeholders of each party to collaborate on defining and managing the decision logic of their blockchain application.

One of the core principles of IBM ODM is bringing together business stakeholders (including policy managers, analysts, lawyers, accountants, auditors) by giving them tools to express and govern the decision logic of their applications in terms they are comfortable with.

Look for an upcoming article that focuses on how you can use Decision Center, coupled with a specific DevOps process, to manage and deploy the logic of smart contracts implemented with IBM ODM.

**Conclusion**

IBM ODM is ideal for expressing and governing the rules in a smart contract. When you define a smart contract, the business stakeholders are necessarily involved, and by separating the rules of the smart contract from the application code, the business stakeholders from both parties can use tools like Decision Center to collaborate to define and govern them. They can also respond quickly to change and avoid the time-consuming processes of updating code bases.

The Vehicle Lifecycle sample illustrates how you can use IBM ODM with Hyperledger Fabric to execute the smart contract decision logic "in-chain" and take advantage of the enterprise-class rule engine of IBM ODM.

For your next blockchain project, consider IBM ODM for your smart contracts. It is a natural fit with Hyperledger Composer and the blockchain environment. The strengths of IBM ODM become particularly important as you bring the business stakeholders closer to the process, and look for ways to quickly respond to change.

We are eager to hear about your experiences and any additional requirements you might have for integrating IBM ODM and blockchain. You can connect with us on Twitter or email, or add a comment at the bottom of this article.
Acknowledgements

Many thanks to the Hyperledger Composer team members for their extraordinary help and responsiveness in supporting this integration. Special thanks to Simon Stone on the IBM Blockchain team for his continuous support.

We also would like to thank Laurent Grateau, Philippe Bonnard, and Jeremy Pichon for their contribution to the sample supporting this article, and Peter Gilliver and Nicolas Sauterey for their attentive reviewing of this article.
Related topics

- Vehicle Lifecycle sample on GitHub that integrates IBM ODM with blockchain: ODMDev/sample-blockchain-vehicle-lifecycle
- IBM Operational Decision Manager product documentation
- Hyperledger Composer
- Hyperledger Fabric
- Vehicle Lifecycle sample for Hyperledger Composer on GitHub: hyperledger/composer-sample-networks
- developerWorks article: Deploy an IBM Operational Decision Manager topology with Docker Compose
- Blog post: Blockchain Smart Contracts: Challenges & Opportunities
- Blockchain journeys on developer.ibm.com/code