**Application design elements**

This section elaborates the key features for client application and smart contract development found in Hyperledger Fabric. A solid understanding of the features will help you design and implement efficient and effective solutions.

* [Contract names](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/contractname.html)
* [Chaincode namespace](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/chaincodenamespace.html)
* [Transaction context](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/transactioncontext.html)
* [Transaction handlers](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/transactionhandler.html)
* [Endorsement policies](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/endorsementpolicies.html)
* [Connection Profile](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionprofile.html)
* [Connection Options](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionoptions.html)
* [Wallet](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/wallet.html)
* [Gateway](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/gateway.html)

**Contract names**

**Audience**: Architects, application and smart contract developers, administrators

A chaincode is a generic container for deploying code to a Hyperledger Fabric blockchain network. One or more related smart contracts are defined within a chaincode. Every smart contract has a name that uniquely identifies it within a chaincode. Applications access a particular smart contract within a chaincode using its contract name.

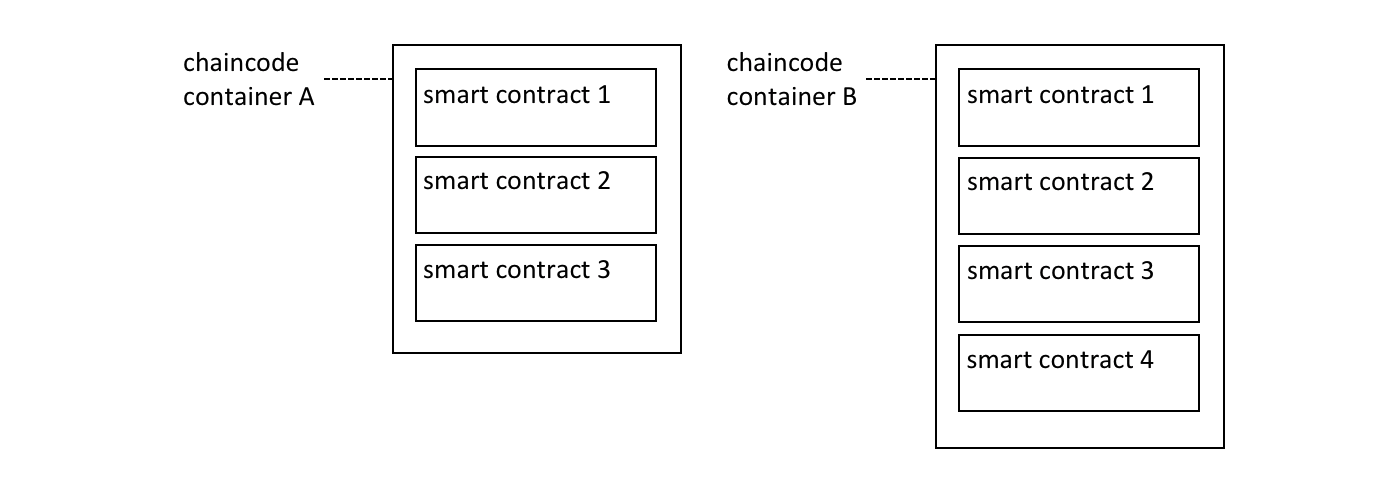
In this topic, we’re going to cover:

* [How a chaincode contains multiple smart contracts](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/contractname.html#chaincode)
* [How to assign a smart contract name](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/contractname.html#name)
* [How to use a smart contract from an application](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/contractname.html#application)
* [The default smart contract](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/contractname.html#default-contract)

**Chaincode**

In the [Developing Applications](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/developing_applications.html) topic, we can see how the Fabric SDKs provide high level programming abstractions which help application and smart contract developers to focus on their business problem, rather than the low level details of how to interact with a Fabric network.

Smart contracts are one example of a high level programming abstraction, and it is possible to define smart contracts within in a chaincode container. When a chaincode is installed on your peer and deployed to a channel, all the smart contracts within it are made available to your applications.

 *Multiple smart contracts can be defined within a chaincode. Each is uniquely identified by their name within a chaincode.*

In the diagram [above](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/contractname.html#chaincode), chaincode A has three smart contracts defined within it, whereas chaincode B has four smart contracts. See how the chaincode name is used to fully qualify a particular smart contract.

The ledger structure is defined by a set of deployed smart contracts. That’s because the ledger contains facts about the business objects of interest to the network (such as commercial paper within PaperNet), and these business objects are moved through their lifecycle (e.g. issue, buy, redeem) by the transaction functions defined within a smart contract.

In most cases, a chaincode will only have one smart contract defined within it. However, it can make sense to keep related smart contracts together in a single chaincode. For example, commercial papers denominated in different currencies might have contracts EuroPaperContract, DollarPaperContract, YenPaperContract which might need to be kept synchronized with each other in the channel to which they are deployed.

**Name**

Each smart contract within a chaincode is uniquely identified by its contract name. A smart contract can explicitly assign this name when the class is constructed, or let the Contract class implicitly assign a default name.

Examine the papercontract.js chaincode [file](https://github.com/hyperledger/fabric-samples/blob/master/commercial-paper/organization/magnetocorp/contract/lib/papercontract.js#L31):

**class** CommercialPaperContract **extends** Contract {

constructor() {

*// Unique name when multiple contracts per chaincode file*

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

}

See how the CommercialPaperContract constructor specifies the contract name as org.papernet.commercialpaper. The result is that within the papercontract chaincode, this smart contract is now associated with the contract name org.papernet.commercialpaper.

If an explicit contract name is not specified, then a default name is assigned – the name of the class. In our example, the default contract name would be CommercialPaperContract.

Choose your names carefully. It’s not just that each smart contract must have a unique name; a well-chosen name is illuminating. Specifically, using an explicit DNS-style naming convention is recommended to help organize clear and meaningful names; org.papernet.commercialpaper conveys that the PaperNet network has defined a standard commercial paper smart contract.

Contract names are also helpful to disambiguate different smart contract transaction functions with the same name in a given chaincode. This happens when smart contracts are closely related; their transaction names will tend to be the same. We can see that a transaction is uniquely defined within a channel by the combination of its chaincode and smart contract name.

Contract names must be unique within a chaincode file. Some code editors will detect multiple definitions of the same class name before deployment. Regardless the chaincode will return an error if multiple classes with the same contract name are explicitly or implicitly specified.

**Application**

Once a chaincode has been installed on a peer and deployed to a channel, the smart contracts in it are accessible to an application:

**const** network **=** await gateway.getNetwork(`papernet`);

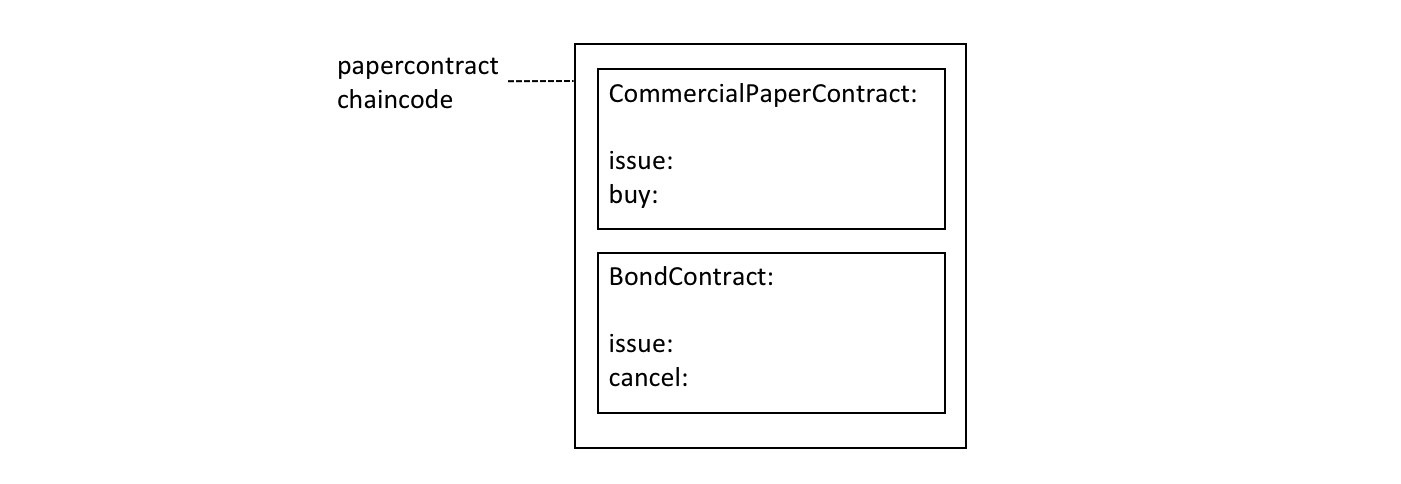
**const** contract **=** await network.getContract('papercontract', 'org.papernet.commercialpaper');

**const** issueResponse **=** await contract.submitTransaction('issue', 'MagnetoCorp', '00001', '2020-05-31', '2020-11-30', '5000000');

See how the application accesses the smart contract with the network.getContract() method. The papercontract chaincode name org.papernet.commercialpaper returns a contract reference which can be used to submit transactions to issue commercial paper with the contract.submitTransaction() API.

**Default contract**

The first smart contract defined in a chaincode is called the *default* smart contract. A default is helpful because a chaincode will usually have one smart contract defined within it; a default allows the application to access those transactions directly – without specifying a contract name.

 *A default smart contract is the first contract defined in a chaincode.*

In this diagram, CommercialPaperContract is the default smart contract. Even though we have two smart contracts, the default smart contract makes our [previous](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/contractname.html#application) example easier to write:

**const** network **=** await gateway.getNetwork(`papernet`);

**const** contract **=** await network.getContract('papercontract');

**const** issueResponse **=** await contract.submitTransaction('issue', 'MagnetoCorp', '00001', '2020-05-31', '2020-11-30', '5000000');

This works because the default smart contract in papercontract is CommercialPaperContract and it has an issue transaction. Note that the issue transaction in BondContract can only be invoked by explicitly addressing it. Likewise, even though the cancel transaction is unique, because BondContract is *not* the default smart contract, it must also be explicitly addressed.

In most cases, a chaincode will only contain a single smart contract, so careful naming of the chaincode can reduce the need for developers to care about chaincode as a concept. In the example code [above](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/contractname.html#default-contract) it feels like papercontract is a smart contract.

In summary, contract names are a straightforward mechanism to identify individual smart contracts within a given chaincode. Contract names make it easy for applications to find a particular smart contract and use it to access the ledger.

**Chaincode namespace**

**Audience**: Architects, application and smart contract developers, administrators

A chaincode namespace allows it to keep its world state separate from other chaincodes. Specifically, smart contracts in the same chaincode share direct access to the same world state, whereas smart contracts in different chaincodes cannot directly access each other’s world state. If a smart contract needs to access another chaincode world state, it can do this by performing a chaincode-to-chaincode invocation. Finally, a blockchain can contain transactions which relate to different world states.

In this topic, we’re going to cover:

* [The importance of namespaces](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/chaincodenamespace.html#motivation)
* [What is a chaincode namespace](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/chaincodenamespace.html#scenario)
* [Channels and namespaces](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/chaincodenamespace.html#channels)
* [How to use chaincode namespaces](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/chaincodenamespace.html#usage)
* [How to access world states across smart contracts](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/chaincodenamespace.html#cross-chaincode-access)
* [Design considerations for chaincode namespaces](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/chaincodenamespace.html#considerations)

**Motivation**

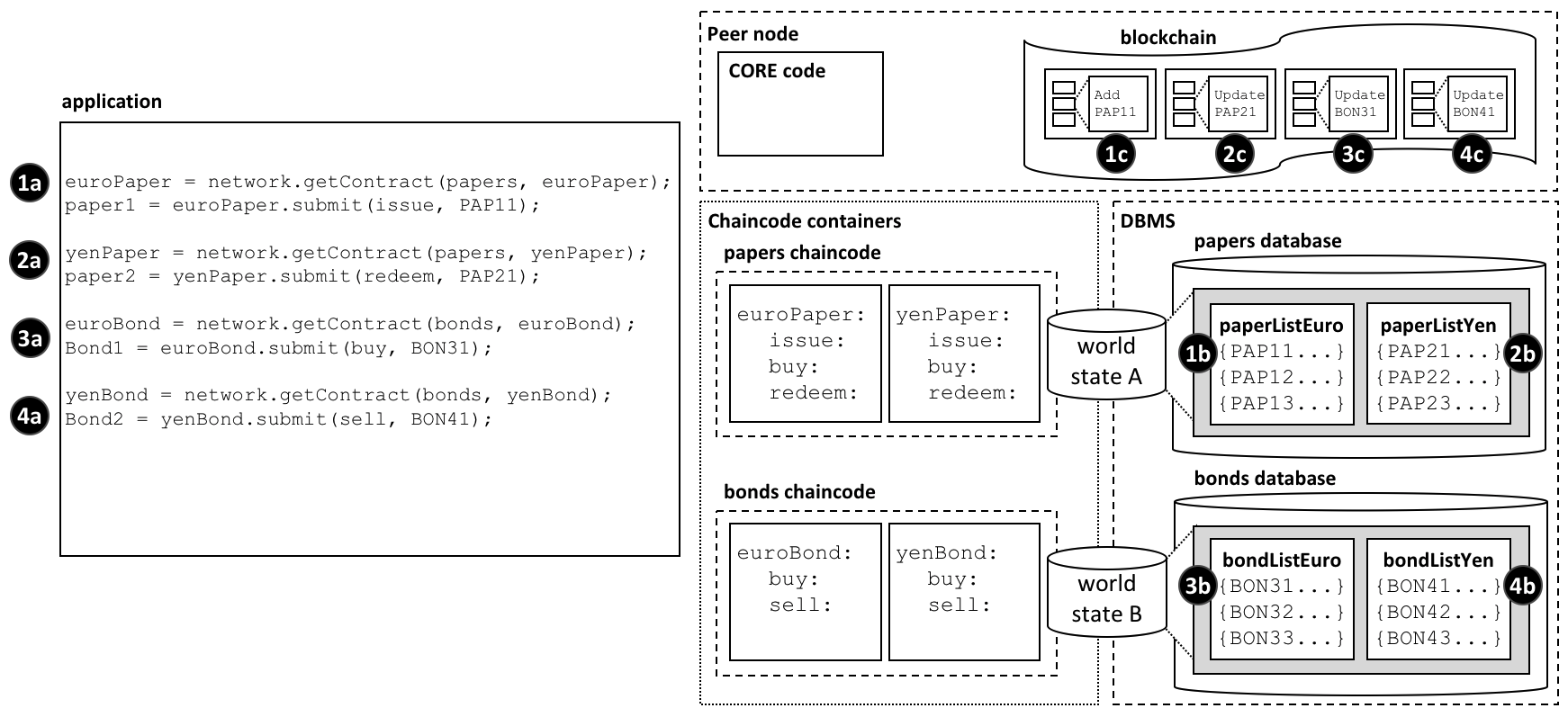
A namespace is a common concept. We understand that *Park Street, New York* and *Park Street, Seattle* are different streets even though they have the same name. The city forms a **namespace** for Park Street, simultaneously providing freedom and clarity.

It’s the same in a computer system. Namespaces allow different users to program and operate different parts of a shared system, without getting in each other’s way. Many programming languages have namespaces so that programs can freely assign unique identifiers, such as variable names, without worrying about other programs doing the same. We’ll see that Hyperledger Fabric uses namespaces to help smart contracts keep their ledger world state separate from other smart contracts.

**Scenario**

Let’s examine how the ledger world state organizes facts about business objects that are important to the organizations in a channel using the diagram below. Whether these objects are commercial papers, bonds, or vehicle registrations, and wherever they are in their lifecycle, they are maintained as states within the ledger world state database. A smart contract manages these business objects by interacting with the ledger (world state and blockchain), and in most cases this will involve it querying or updating the ledger world state.

It’s vitally important to understand that the ledger world state is partitioned according to the chaincode of the smart contract that accesses it, and this partitioning, or *namespacing* is an important design consideration for architects, administrators and programmers.

 *The ledger world state is separated into different namespaces according to the chaincode that accesses it. Within a given channel, smart contracts in the same chaincode share the same world state, and smart contracts in different chaincodes cannot directly access each other’s world state. Likewise, a blockchain can contain transactions that relate to different chaincode world states.*

In our example, we can see four smart contracts defined in two different chaincodes, each of which is in their own chaincode container. The euroPaper and yenPaper smart contracts are defined in the papers chaincode. The situation is similar for the euroBond and yenBond smart contracts – they are defined in the bonds chaincode. This design helps application programmers understand whether they are working with commercial papers or bonds priced in Euros or Yen, and because the rules for each financial product don’t really change for different currencies, it makes sense to manage their deployment in the same chaincode.

The [diagram](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/chaincodenamespace.html#scenario) also shows the consequences of this deployment choice. The database management system (DBMS) creates different world state databases for the papers and bonds chaincodes and the smart contracts contained within them. World state A and world state B are each held within distinct databases; the data are isolated from each other such that a single world state query (for example) cannot access both world states. The world state is said to be *namespaced* according to its chaincode.

See how world state A contains two lists of commercial papers paperListEuro and paperListYen. The states PAP11 and PAP21 are instances of each paper managed by the euroPaper and yenPaper smart contracts respectively. Because they share the same chaincode namespace, their keys (PAPxyz) must be unique within the namespace of the papers chaincode, a little like a street name is unique within a town. Notice how it would be possible to write a smart contract in the papers chaincode that performed an aggregate calculation over all the commercial papers – whether priced in Euros or Yen – because they share the same namespace. The situation is similar for bonds – they are held within world state B which maps to a separate bonds database, and their keys must be unique.

Just as importantly, namespaces mean that euroPaper and yenPaper cannot directly access world state B, and that euroBond and yenBond cannot directly access world state A. This isolation is helpful, as commercial papers and bonds are very distinct financial instruments; they have different attributes and are subject to different rules. It also means that papers and bonds could have the same keys, because they are in different namespaces. This is helpful; it provides a significant degree of freedom for naming. Use this freedom to name different business objects meaningfully.

Most importantly, we can see that a blockchain is associated with the peer operating in a particular channel, and that it contains transactions that affect both world state A and world state B. That’s because the blockchain is the most fundamental data structure contained in a peer. The set of world states can always be recreated from this blockchain, because they are the cumulative results of the blockchain’s transactions. A world state helps simplify smart contracts and improve their efficiency, as they usually only require the current value of a state. Keeping world states separate via namespaces helps smart contracts isolate their logic from other smart contracts, rather than having to worry about transactions that correspond to different world states. For example, a bonds contract does not need to worry about paper transactions, because it cannot see their resultant world state.

It’s also worth noticing that the peer, chaincode containers and DBMS all are logically different processes. The peer and all its chaincode containers are always in physically separate operating system processes, but the DBMS can be configured to be embedded or separate, depending on its [type](https://hyperledger-fabric.readthedocs.io/en/latest/ledger/ledger.html#world-state-database-options). For LevelDB, the DBMS is wholly contained within the peer, but for CouchDB, it is a separate operating system process.

It’s important to remember that namespace choices in this example are the result of a business requirement to share commercial papers in different currencies but isolate them separate from bonds. Think about how the namespace structure would be modified to meet a business requirement to keep every financial asset class separate, or share all commercial papers and bonds?

**Channels**

If a peer is joined to multiple channels, then a new blockchain is created and managed for each channel. Moreover, every time a chaincode is deployed to a new channel, a new world state database is created for it. It means that the channel also forms a kind of namespace alongside that of the chaincode for the world state.

However, the same peer and chaincode container processes can be simultaneously joined to multiple channels – unlike blockchains, and world state databases, these processes do not increase with the number of channels joined.

For example, if you deployed the papers and bonds chaincode to a new channel, there would a totally separate blockchain created, and two new world state databases created. However, the peer and chaincode containers would not increase; each would just be connected to multiple channels.

**Usage**

Let’s use our commercial paper [example](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/chaincodenamespace.html#scenario) to show how an application uses a smart contract with namespaces. It’s worth noting that an application communicates with the peer, and the peer routes the request to the appropriate chaincode container which then accesses the DBMS. This routing is done by the peer **core** component shown in the diagram.

Here’s the code for an application that uses both commercial papers and bonds, priced in Euros and Yen. The code is fairly self-explanatory:

**const** euroPaper **=** network.getContract(papers, euroPaper);

paper1 **=** euroPaper.submit(issue, PAP11);

**const** yenPaper **=** network.getContract(papers, yenPaper);

paper2 **=** yenPaper.submit(redeem, PAP21);

**const** euroBond **=** network.getContract(bonds, euroBond);

bond1 **=** euroBond.submit(buy, BON31);

**const** yenBond **=** network.getContract(bonds, yenBond);

bond2 **=** yenBond.submit(sell, BON41);

See how the application:

* Accesses the euroPaper and yenPaper contracts using the getContract() API specifying the papers chaincode. See interaction points **1a** and **2a**.
* Accesses the euroBond and yenBond contracts using the getContract() API specifying the bonds chaincode. See interaction points **3a** and **4a**.
* Submits an issue transaction to the network for commercial paper PAP11 using the euroPaper contract. See interaction point **1a**. This results in the creation of a commercial paper represented by state PAP11 in world state A; interaction point **1b**. This operation is captured as a transaction in the blockchain at interaction point **1c**.
* Submits a redeem transaction to the network for commercial paper PAP21 using the yenPaper contract. See interaction point **2a**. This results in the creation of a commercial paper represented by state PAP21 in world state A; interaction point **2b**. This operation is captured as a transaction in the blockchain at interaction point **2c**.
* Submits a buy transaction to the network for bond BON31 using the euroBond contract. See interaction point **3a**. This results in the creation of a bond represented by state BON31 in world state B; interaction point **3b**. This operation is captured as a transaction in the blockchain at interaction point **3c**.
* Submits a sell transaction to the network for bond BON41 using the yenBond contract. See interaction point **4a**. This results in the creation of a bond represented by state BON41 in world state B; interaction point **4b**. This operation is captured as a transaction in the blockchain at interaction point **4c**.

See how smart contracts interact with the world state:

* euroPaper and yenPaper contracts can directly access world state A, but cannot directly access world state B. World state A is physically held in the papers database in the database management system (DBMS) corresponding to the papers chaincode.
* euroBond and yenBond contracts can directly access world state B, but cannot directly access world state A. World state B is physically held in the bonds database in the database management system (DBMS) corresponding to the bonds chaincode.

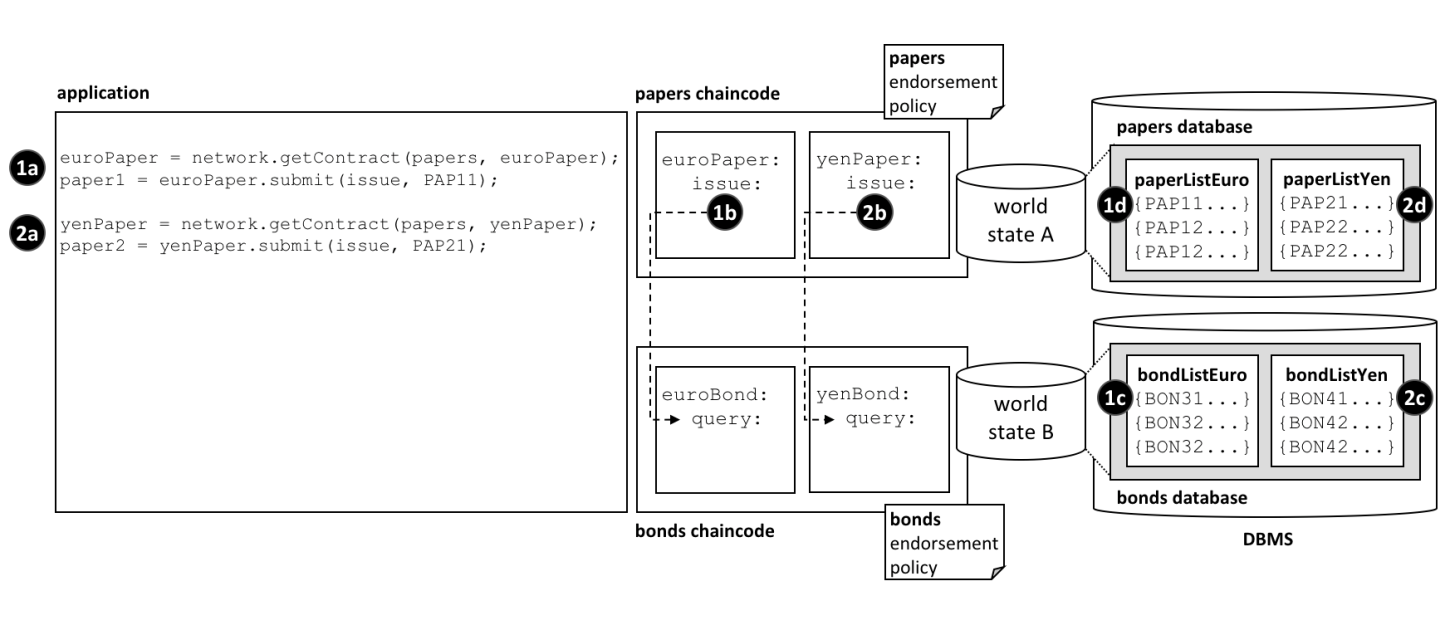
See how the blockchain captures transactions for all world states:

* Interactions **1c** and **2c** correspond to transactions create and update commercial papers PAP11 and PAP21 respectively. These are both contained within world state A.
* Interactions **3c** and **4c** correspond to transactions both update bonds BON31 and BON41. These are both contained within world state B.
* If world state A or world state B were destroyed for any reason, they could be recreated by replaying all the transactions in the blockchain.

**Cross chaincode access**

As we saw in our example [scenario](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/chaincodenamespace.html#scenario), euroPaper and yenPaper cannot directly access world state B. That’s because we have designed our chaincodes and smart contracts so that these chaincodes and world states are kept separately from each other. However, let’s imagine that euroPaper needs to access world state B.

Why might this happen? Imagine that when a commercial paper was issued, the smart contract wanted to price the paper according to the current return on bonds with a similar maturity date. In this case it will be necessary for the euroPaper contract to be able to query the price of bonds in world state B. Look at the following diagram to see how we might structure this interaction.

 *How chaincodes and smart contracts can indirectly access another world state – via its chaincode.*

Notice how:

* the application submits an issue transaction in the euroPaper smart contract to issue PAP11. See interaction **1a**.
* the issue transaction in the euroPaper smart contract calls the query transaction in the euroBond smart contract. See interaction point **1b**.
* the queryin euroBond can retrieve information from world state B. See interaction point **1c**.
* when control returns to the issue transaction, it can use the information in the response to price the paper and update world state A with information. See interaction point **1d**.
* the flow of control for issuing commercial paper priced in Yen is the same. See interaction points **2a**, **2b**, **2c** and **2d**.

Control is passed between chaincode using the invokeChaincode() [API](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#invokeChaincode__anchor). This API passes control from one chaincode to another chaincode.

Although we have only discussed query transactions in the example, it is possible to invoke a smart contract which will update the called chaincode’s world state. See the [considerations](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/chaincodenamespace.html#considerations) below.

**Considerations**

* In general, each chaincode will have a single smart contract in it.
* Multiple smart contracts should only be deployed in the same chaincode if they are very closely related. Usually, this is only necessary if they share the same world state.
* Chaincode namespaces provide isolation between different world states. In general it makes sense to isolate unrelated data from each other. Note that you cannot choose the chaincode namespace; it is assigned by Hyperledger Fabric, and maps directly to the name of the chaincode.
* For chaincode to chaincode interactions using the invokeChaincode() API, both chaincodes must be installed on the same peer.
  + For interactions that only require the called chaincode’s world state to be queried, the invocation can be in a different channel to the caller’s chaincode.
  + For interactions that require the called chaincode’s world state to be updated, the invocation must be in the same channel as the caller’s chaincode.

**Transaction context**

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

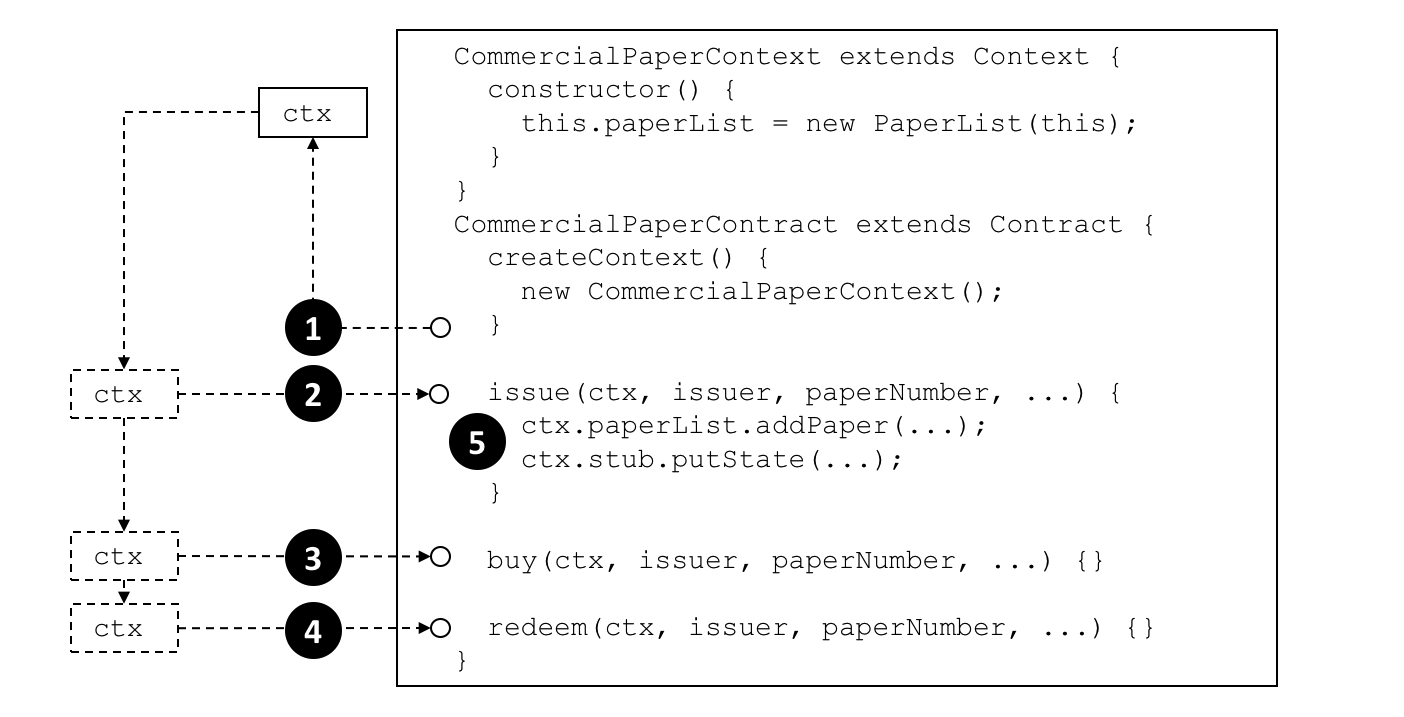
A transaction context performs two functions. Firstly, it allows a developer to define and maintain user variables across transaction invocations within a smart contract. Secondly, it provides access to a wide range of Fabric APIs that allow smart contract developers to perform operations relating to detailed transaction processing. These range from querying or updating the ledger, both the immutable blockchain and the modifiable world state, to retrieving the transaction-submitting application’s digital identity.

A transaction context is created when a smart contract is deployed to a channel and made available to every subsequent transaction invocation. A transaction context helps smart contract developers write programs that are powerful, efficient and easy to reason about.

* [Why a transaction context is important](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/transactioncontext.html#scenario)
* [How to use a transaction context](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/transactioncontext.html#programming)
* [What’s in a transaction context](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/transactioncontext.html#structure)
* [Using a context stub](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/transactioncontext.html#stub)
* [Using a context clientIdentity](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/transactioncontext.html#clientIdentity)

**Scenario**

In the commercial paper sample, [papercontract](https://github.com/hyperledger/fabric-samples/blob/master/commercial-paper/organization/magnetocorp/contract/lib/papercontract.js) initially defines the name of the list of commercial papers for which it’s responsible. Each transaction subsequently refers to this list; the issue transaction adds new papers to it, the buy transaction changes its owner, and the redeem transaction marks it as complete. This is a common pattern; when writing a smart contract it’s often helpful to initialize and recall particular variables in sequential transactions.

 *A smart contract transaction context allows smart contracts to define and maintain user variables across transaction invocations. Refer to the text for a detailed explanation.*

**Programming**

When a smart contract is constructed, a developer can optionally override the built-in Context class createContext method to create a custom context:

createContext() {

**new** CommercialPaperContext();

}

In our example, the CommercialPaperContext is specialized for CommercialPaperContract. See how the custom context, addressed through this, adds the specific variable PaperList to itself:

CommercialPaperContext **extends** Context {

constructor () {

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

}

}

When the createContext() method returns at point **(1)** in the diagram [above](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/transactioncontext.html#scenario), a custom context ctx has been created which contains paperList as one of its variables.

Subsequently, whenever a smart contract transaction such as issue, buy or redeem is called, this context will be passed to it. See how at points **(2)**, **(3)** and **(4)** the same commercial paper context is passed into the transaction method using the ctx variable.

See how the context is then used at point **(5)**:

ctx.paperList.addPaper(...);

ctx.stub.putState(...);

Notice how paperList created in CommercialPaperContext is available to the issue transaction. See how paperList is similarly used by the **redeem** and **buy** transactions; ctx makes the smart contracts efficient and easy to reason about.

You can also see that there’s another element in the context – ctx.stub – which was not explictly added by CommercialPaperContext. That’s because stub and other variables are part of the built-in context. Let’s now examine the structure of this built-in context, these implicit variables and how to use them.

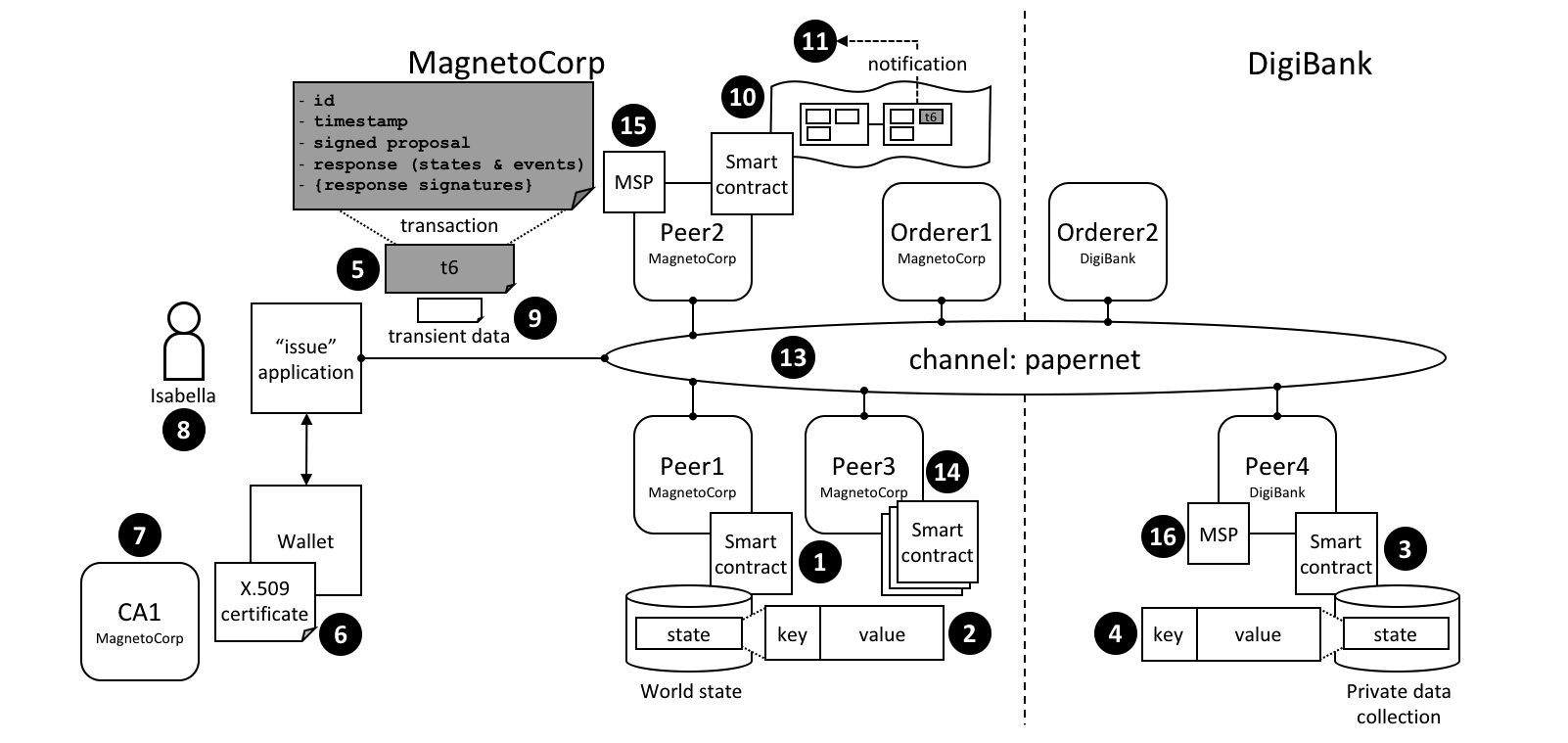
**Structure**

As we’ve seen from the [example](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/transactioncontext.html#programming), a transaction context can contain any number of user variables such as paperList.

The transaction context also contains two built-in elements that provide access to a wide range of Fabric functionality ranging from the client application that submitted the transaction to ledger access.

* ctx.stub is used to access APIs that provide a broad range of transaction processing operations from putState() and getState() to access the ledger, to getTxID() to retrieve the current transaction ID.
* ctx.clientIdentity is used to get information about the identity of the user who submitted the transaction.

We’ll use the following diagram to show you what a smart contract can do using the stub and clientIdentity using the APIs available to it:

 *A smart contract can access a range of functionality in a smart contract via the transaction context stub and clientIdentity. Refer to the text for a detailed explanation.*

**Stub**

The APIs in the stub fall into the following categories:

* **World state data APIs**. See interaction point **(1)**. These APIs enable smart contracts to get, put and delete state corresponding to individual objects from the world state, using their key:
  + [getState()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getState__anchor)
  + [putState()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#putState__anchor)
  + [deleteState()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#deleteState__anchor)

These basic APIs are complemented by query APIs which enable contracts to retrieve a set of states, rather than an individual state. See interaction point **(2)**. The set is either defined by a range of key values, using full or partial keys, or a query according to values in the underlying world state [database](https://hyperledger-fabric.readthedocs.io/en/latest/ledger/ledger.html#world-state-database-options). For large queries, the result sets can be paginated to reduce storage requirements:

* + [getStateByRange()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getStateByRange__anchor)
  + [getStateByRangeWithPagination()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getStateByRangeWithPagination__anchor)
  + [getStateByPartialCompositeKey()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getStateByPartialCompositeKey__anchor)
  + [getStateByPartialCompositeKeyWithPagination()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getStateByPartialCompositeKeyWithPagination__anchor)
  + [getQueryResult()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getQueryResult__anchor)
  + [getQueryResultWithPagination()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getQueryResultWithPagination__anchor)
* **Private data APIs**. See interaction point **(3)**. These APIs enable smart contracts to interact with a private data collection. They are analogous to the APIs for world state interactions, but for private data. There are APIs to get, put and delete a private data state by its key:
  + [getPrivateData()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getPrivateData__anchor)
  + [putPrivateData()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#putPrivateData__anchor)
  + [deletePrivateData()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#deletePrivateData__anchor)

This set is complemented by set of APIs to query private data **(4)**. These APIs allow smart contracts to retrieve a set of states from a private data collection, according to a range of key values, either full or partial keys, or a query according to values in the underlying world state [database](https://hyperledger-fabric.readthedocs.io/en/latest/ledger/ledger.html#world-state-database-options). There are currently no pagination APIs for private data collections.

* + [getPrivateDataByRange()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getPrivateDataByRange__anchor)
  + [getPrivateDataByPartialCompositeKey()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getPrivateDataByPartialCompositeKey__anchor)
  + [getPrivateDataQueryResult()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getPrivateDataQueryResult__anchor)
* **Transaction APIs**. See interaction point **(5)**. These APIs are used by a smart contract to retrieve details about the current transaction proposal being processed by the smart contract. This includes the transaction identifier and the time when the transaction proposal was created.
  + [getTxID()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getTxID__anchor) returns the identifier of the current transaction proposal **(5)**.
  + [getTxTimestamp()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getTxTimestamp__anchor) returns the timestamp when the current transaction proposal was created by the application **(5)**.
  + [getCreator()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getCreator__anchor) returns the raw identity (X.509 or otherwise) of the creator of transaction proposal. If this is an X.509 certificate then it is often more appropriate to use [ctx.ClientIdentity](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/transactioncontext.html" \l "clientidentity).
  + [getSignedProposal()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getSignedProposal__anchor) returns a signed copy of the current transaction proposal being processed by the smart contract.
  + [getBinding()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getBinding__anchor) is used to prevent transactions being maliciously or accidentally replayed using a nonce. (For practical purposes, a nonce is a random number generated by the client application and incorporated in a cryptographic hash.) For example, this API could be used by a smart contract at **(1)** to detect a replay of the transaction **(5)**.
  + [getTransient()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getTransient__anchor) allows a smart contract to access the transient data an application passes to a smart contract. See interaction points **(9)** and **(10)**. Transient data is private to the application-smart contract interaction. It is not recorded on the ledger and is often used in conjunction with private data collections **(3)**.
* **Key APIs** are used by smart contracts to manipulate state key in the world state or a private data collection. See interaction points **2** and **4**.

The simplest of these APIs allows smart contracts to form and split composite keys from their individual components. Slightly more advanced are the ValidationParameter() APIs which get and set the state based endorsement policies for world state **(2)** and private data **(4)**. Finally, getHistoryForKey() retrieves the history for a state by returning the set of stored values, including the transaction identifiers that performed the state update, allowing the transactions to be read from the blockchain **(10)**.

* + [createCompositeKey()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#createCompositeKey__anchor)
  + [splitCompositeKey()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#splitCompositeKey__anchor)
  + [setStateValidationParameter()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#setStateValidationParameter__anchor)
  + [getStateValidationParameter()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getStateValidationParameter__anchor)
  + [getPrivateDataValidationParameter()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getPrivateDataValidationParameter__anchor)
  + [setPrivateDataValidationParameter()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#setPrivateDataValidationParameter__anchor)
  + [getHistoryForKey()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getHistoryForKey__anchor)
* **Event APIs** are used manage event processing in a smart contract.
  + [setEvent()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#setEvent__anchor)

Smart contracts use this API to add user events to a transaction response. See interaction point **(5)**. These events are ultimately recorded on the blockchain and sent to listening applications at interaction point **(11)**.

* **Utility APIs** are a collection of useful APIs that don’t easily fit in a pre-defined category, so we’ve grouped them together! They include retrieving the current channel name and passing control to a different chaincode on the same peer.
  + [getChannelID()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getChannelID__anchor)

See interaction point **(13)**. A smart contract running on any peer can use this API to determined on which channel the application invoked the smart contract.

* + [invokeChaincode()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#invokeChaincode__anchor)

See interaction point **(14)**. Peer3 owned by MagnetoCorp has multiple smart contracts installed on it. These smart contracts are able to call each other using this API. The smart contracts must be collocated; it is not possible to call a smart contract on a different peer.

* Some of these utility APIs are only used if you’re using low-level chaincode, rather than smart contracts. These APIs are primarily for the detailed manipulation of chaincode input; the smart contract Contract class does all of this parameter marshalling automatically for developers.
  + [getFunctionAndParameters()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getFunctionAndParameters__anchor)
  + [getStringArgs()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getStringArgs__anchor)
  + [getArgs()](https://fabric-shim.github.io/master/fabric-shim.ChaincodeStub.html#getArgs__anchor)

**ClientIdentity**

In most cases, the application submitting a transaction will be using an X.509 certificate. In the [example](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/transactioncontext.html#structure), an X.509 certificate **(6)** issued by CA1 **(7)** is being used by Isabella **(8)** in her application to sign the proposal in transaction t6 **(5)**.

ClientIdentity takes the information returned by getCreator() and puts a set of X.509 utility APIs on top of it to make it easier to use for this common use case.

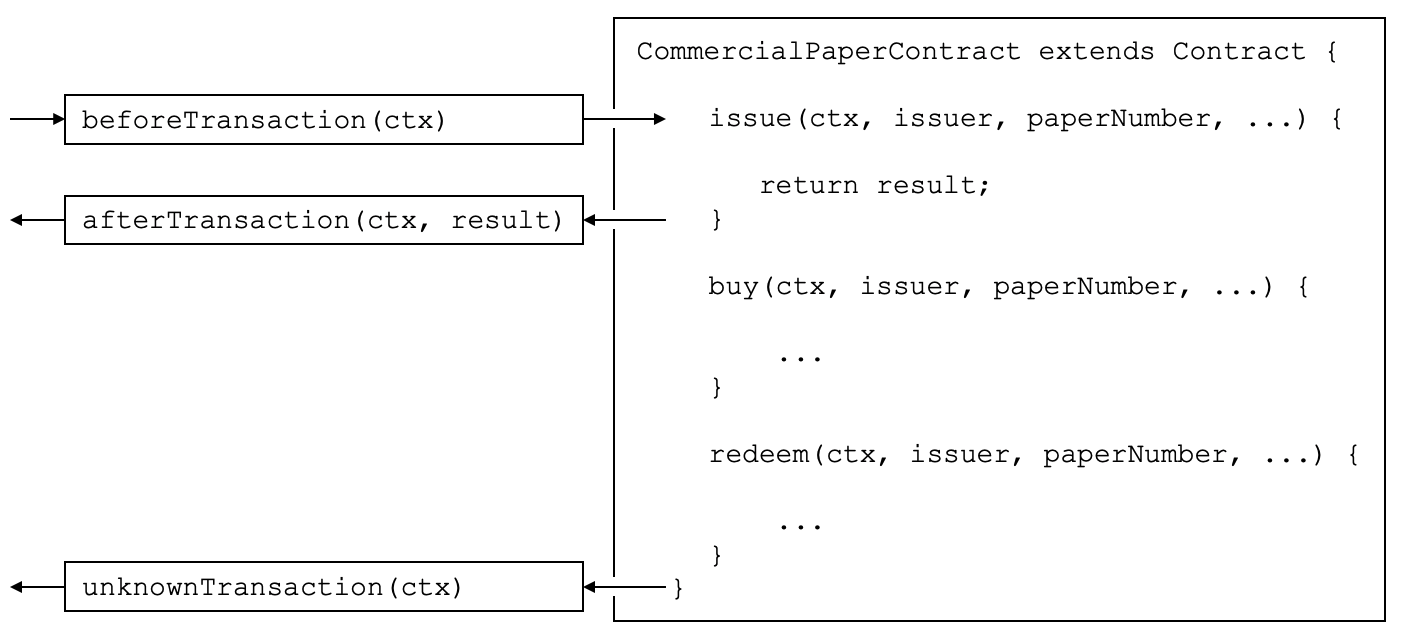
* [getX509Certificate()](https://fabric-shim.github.io/master/fabric-shim.ClientIdentity.html#getX509Certificate__anchor) returns the full X.509 certificate of the transaction submitter, including all its attributes and their values. See interaction point **(6)**.
* [getAttributeValue()](https://fabric-shim.github.io/master/fabric-shim.ClientIdentity.html#getAttributeValue__anchor) returns the value of a particular X.509 attribute, for example, the organizational unit OU, or distinguished name DN. See interaction point **(6)**.
* [assertAttributeValue()](https://fabric-shim.github.io/master/fabric-shim.ClientIdentity.html#assertAttributeValue__anchor) returns TRUE if the specified attribute of the X.509 attribute has a specified value. See interaction point **(6)**.
* [getID()](https://fabric-shim.github.io/master/fabric-shim.ClientIdentity.html#getID__anchor) returns the unique identity of the transaction submitter, according to their distinguished name and the issuing CA’s distinguished name. The format is x509::{subject DN}::{issuer DN}. See interaction point **(6)**.
* [getMSPID()](https://fabric-shim.github.io/master/fabric-shim.ClientIdentity.html#getMSPID__anchor) returns the channel MSP of the transaction submitter. This allows a smart contract to make processing decisions based on the submitter’s organizational identity. See interaction point **(15)** or **(16)**.

**Transaction handlers**

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

Transaction handlers allow smart contract developers to define common processing at key points during the interaction between an application and a smart contract. Transaction handlers are optional but, if defined, they will receive control before or after every transaction in a smart contract is invoked. There is also a specific handler which receives control when a request is made to invoke a transaction not defined in a smart contract.

Here’s an example of transaction handlers for the [commercial paper smart contract sample](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/smartcontract.html):



*Before, After and Unknown transaction handlers. In this example, beforeTransaction() is called before the****issue****,****buy****and****redeem****transactions. afterTransaction() is called after the****issue****,****buy****and****redeem****transactions. unknownTransaction() is only called if a request is made to invoke a transaction not defined in the smart contract. (The diagram is simplified by not repeating beforeTransaction and afterTransaction boxes for each transaction.)*

**Types of handler**

There are three types of transaction handlers which cover different aspects of the interaction between an application and a smart contract:

* **Before handler**: is called before every smart contract transaction is invoked. The handler will usually modify the transaction context to be used by the transaction. The handler has access to the full range of Fabric APIs; for example, it can issue getState() and putState().
* **After handler**: is called after every smart contract transaction is invoked. The handler will usually perform post-processing common to all transactions, and also has full access to the Fabric APIs.
* **Unknown handler**: is called if an attempt is made to invoke a transaction that is not defined in a smart contract. Typically, the handler will record the failure for subsequent processing by an administrator. The handler has full access to the Fabric APIs.

Defining a transaction handler is optional; a smart contract will perform correctly without handlers being defined. A smart contract can define at most one handler of each type.

**Defining a handler**

Transaction handlers are added to the smart contract as methods with well defined names. Here’s an example which adds a handler of each type:

CommercialPaperContract **extends** Contract {

...

async beforeTransaction(ctx) {

*// Write the transaction ID as an informational to the console*

console.info(ctx.stub.getTxID());

};

async afterTransaction(ctx, result) {

*// This handler interacts with the ledger*

ctx.stub.cpList.putState(...);

};

async unknownTransaction(ctx) {

*// This handler throws an exception*

**throw** **new** Error('Unknown transaction function');

};

}

The form of a transaction handler definition is the similar for all handler types, but notice how the afterTransaction(ctx, result) also receives any result returned by the transaction. The [API documentation](https://fabric-shim.github.io/release-1.4/fabric-contract-api.Contract.html) shows you the exact form of these handlers.

**Handler processing**

Once a handler has been added to the smart contract, it will be invoked during transaction processing. During processing, the handler receives ctx, the [transaction context](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/transationcontext.md), performs some processing, and returns control as it completes. Processing continues as follows:

* **Before handler**: If the handler completes successfully, the transaction is called with the updated context. If the handler throws an exception, then the transaction is not called and the smart contract fails with the exception error message.
* **After handler**: If the handler completes successfully, then the smart contract completes as determined by the invoked transaction. If the handler throws an exception, then the transaction fails with the exception error message.
* **Unknown handler**: The handler should complete by throwing an exception with the required error message. If an **Unknown handler** is not specified, or an exception is not thrown by it, there is sensible default processing; the smart contract will fail with an **unknown transaction** error message.

If the handler requires access to the function and parameters, then it is easy to do this:

async beforeTransaction(ctx) {

*// Retrieve details of the transaction*

**let** txnDetails **=** ctx.stub.getFunctionAndParameters();

console.info(`Calling function: ${txnDetails.fcn} `);

console.info(util.format(`Function arguments : %j ${stub.getArgs()} ``);

}

See how this handler uses the utility API getFunctionAndParameters via the [transaction context](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/transactioncontext.html#stub).

**Multiple handlers**

It is only possible to define at most one handler of each type for a smart contract. If a smart contract needs to invoke multiple functions during before, after or unknown handling, it should coordinate this from within the appropriate function.

# Endorsement policies

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

Endorsement policies define the smallest set of organizations that are required to endorse a transaction in order for it to be valid. To endorse, an organization’s endorsing peer needs to run the smart contract associated with the transaction and sign its outcome. When the ordering service sends the transaction to the committing peers, they will each individually check whether the endorsements in the transaction fulfill the endorsement policy. If this is not the case, the transaction is invalidated and it will have no effect on world state.

Endorsement policies work at two different granularities: they can be set for an entire namespace, as well as for individual state keys. They are formulated using basic logic expressions such as AND and OR. For example, in PaperNet this could be used as follows: the endorsement policy for a paper that has been sold from MagnetoCorp to DigiBank could be set to AND(MagnetoCorp.peer, DigiBank.peer), requiring any changes to this paper to be endorsed by both MagnetoCorp and DigiBank.

**Connection Profile**

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

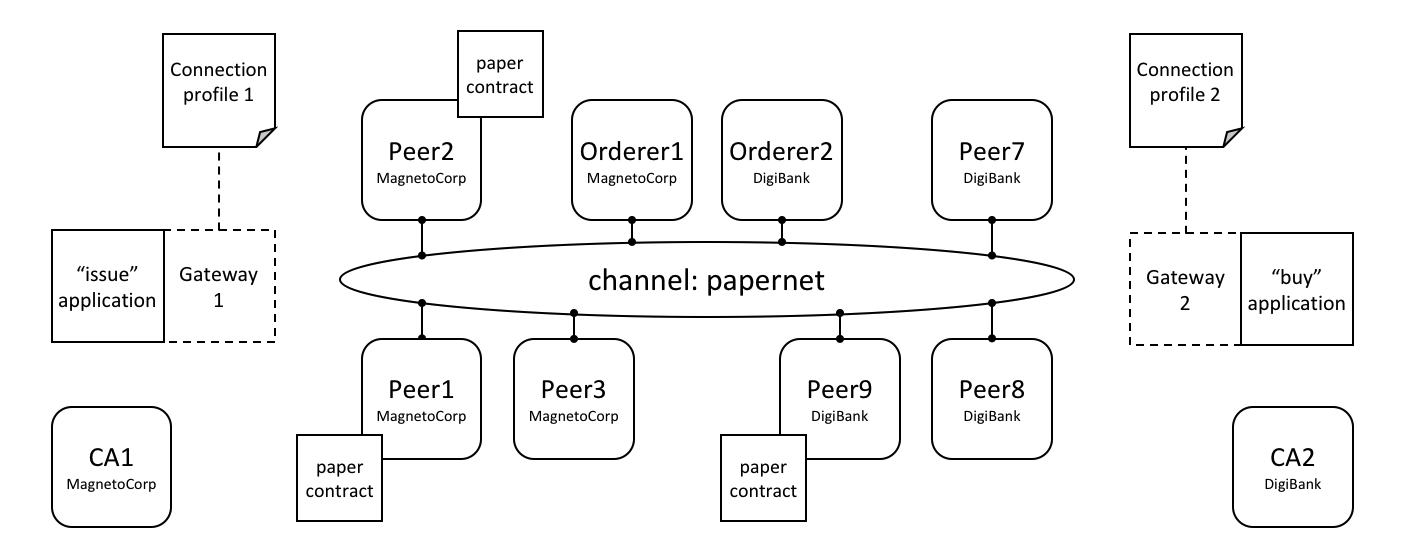
A connection profile describes a set of components, including peers, orderers and certificate authorities in a Hyperledger Fabric blockchain network. It also contains channel and organization information relating to these components. A connection profile is primarily used by an application to configure a [gateway](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/gateway.html) that handles all network interactions, allowing it to focus on business logic. A connection profile is normally created by an administrator who understands the network topology.

In this topic, we’re going to cover:

* [Why connection profiles are important](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionprofile.html#scenario)
* [How applications use a connection profile](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionprofile.html#usage)
* [How to define a connection profile](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionprofile.html#structure)

**Scenario**

A connection profile is used to configure a gateway. Gateways are important for [many reasons](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/gateway.html), the primary being to simplify an application’s interaction with a network channel.

 *Two applications, issue and buy, use gateways 1&2 configured with connection profiles 1&2. Each profile describes a different subset of MagnetoCorp and DigiBank network components. Each connection profile must contain sufficient information for a gateway to interact with the network on behalf of the issue and buy applications. See the text for a detailed explanation.*

A connection profile contains a description of a network view, expressed in a technical syntax, which can either be JSON or YAML. In this topic, we use the YAML representation, as it’s easier for you to read. Static gateways need more information than dynamic gateways because the latter can use [service discovery](https://hyperledger-fabric.readthedocs.io/en/latest/discovery-overview.html) to dynamically augment the information in a connection profile.

A connection profile should not be an exhaustive description of a network channel; it just needs to contain enough information sufficient for a gateway that’s using it. In the network above, connection profile 1 needs to contain at least the endorsing organizations and peers for the issue transaction, as well as identifying the peers that will notify the gateway when the transaction has been committed to the ledger.

It’s easiest to think of a connection profile as describing a *view* of the network. It could be a comprehensive view, but that’s unrealistic for a few reasons:

* Peers, orderers, certificate authorities, channels, and organizations are added and removed according to demand.
* Components can start and stop, or fail unexpectedly (e.g. power outage).
* A gateway doesn’t need a view of the whole network, only what’s necessary to successfully handle transaction submission or event notification for example.
* Service Discovery can augment the information in a connection profile. Specifically, dynamic gateways can be configured with minimal Fabric topology information; the rest can be discovered.

A static connection profile is normally created by an administrator who understands the network topology in detail. That’s because a static profile can contain quite a lot of information, and an administrator needs to capture this in the corresponding connection profile. In contrast, dynamic profiles minimize the amount of definition required and therefore can be a better choice for developers who want to get going quickly, or administrators who want to create a more responsive gateway. Connection profiles are created in either the YAML or JSON format using an editor of choice.

**Usage**

We’ll see how to define a connection profile in a moment; let’s first see how it is used by a sample MagnetoCorp issue application:

**const** yaml **=** require('js-yaml');

**const** { Gateway } **=** require('fabric-network');

**const** connectionProfile **=** yaml.safeLoad(fs.readFileSync('../gateway/paperNet.yaml', 'utf8'));

**const** gateway **=** **new** Gateway();

await gateway.connect(connectionProfile, connectionOptions);

After loading some required classes, see how the paperNet.yaml gateway file is loaded from the file system, converted to a JSON object using the yaml.safeLoad() method, and used to configure a gateway using its connect() method.

By configuring a gateway with this connection profile, the issue application is providing the gateway with the relevant network topology it should use to process transactions. That’s because the connection profile contains sufficient information about the PaperNet channels, organizations, peers, orderers and CAs to ensure transactions can be successfully processed.

It’s good practice for a connection profile to define more than one peer for any given organization – it prevents a single point of failure. This practice also applies to dynamic gateways; to provide more than one starting point for service discovery.

A DigiBank buy application would typically configure its gateway with a similar connection profile, but with some important differences. Some elements will be the same, such as the channel; some elements will overlap, such as the endorsing peers. Other elements will be completely different, such as notification peers or certificate authorities for example.

The connectionOptions passed to a gateway complement the connection profile. They allow an application to declare how it would like the gateway to use the connection profile. They are interpreted by the SDK to control interaction patterns with network components, for example to select which identity to connect with, or which peers to use for event notifications. Read [about](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectoptions.html) the list of available connection options and when to use them.

**Structure**

To help you understand the structure of a connection profile, we’re going to step through an example for the network shown [above](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionprofile.html#scenario). Its connection profile is based on the PaperNet commercial paper sample, and [stored](https://github.com/hyperledger/fabric-samples/blob/master/commercial-paper/organization/magnetocorp/gateway/networkConnection.yaml) in the GitHub repository. For convenience, we’ve reproduced it [below](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionprofile.html#sample). You will find it helpful to display it in another browser window as you now read about it:

* Line 9: name: "papernet.magnetocorp.profile.sample"

This is the name of the connection profile. Try to use DNS style names; they are a very easy way to convey meaning.

* Line 16: x-type: "hlfv1"

Users can add their own x- properties that are “application-specific” – just like with HTTP headers. They are provided primarily for future use.

* Line 20: description: "Sample connection profile for documentation topic"

A short description of the connection profile. Try to make this helpful for the reader who might be seeing this for the first time!

* Line 25: version: "1.0"

The schema version for this connection profile. Currently only version 1.0 is supported, and it is not envisioned that this schema will change frequently.

* Line 32: channels:

This is the first really important line. channels: identifies that what follows are *all* the channels that this connection profile describes. However, it is good practice to keep different channels in different connection profiles, especially if they are used independently of each other.

* Line 36: papernet:

Details of papernet, the first channel in this connection profile, will follow.

* Line 41: orderers:

Details of all the orderers for papernet follow. You can see in line 45 that the orderer for this channel is orderer1.magnetocorp.example.com. This is just a logical name; later in the connection profile (lines 134 - 147), there will be details of how to connect to this orderer. Notice that orderer2.digibank.example.com is not in this list; it makes sense that applications use their own organization’s orderers, rather than those from a different organization.

* Line 49: peers:

Details of all the peers for papernet will follow.

You can see three peers listed from MagnetoCorp: peer1.magnetocorp.example.com, peer2.magnetocorp.example.com and peer3.magnetocorp.example.com. It’s not necessary to list all the peers in MagnetoCorp, as has been done here. You can see only one peer listed from DigiBank: peer9.digibank.example.com; including this peer starts to imply that the endorsement policy requires MagnetoCorp and DigiBank to endorse transactions, as we’ll now confirm. It’s good practice to have multiple peers to avoid single points of failure.

Underneath each peer you can see four non-exclusive roles: **endorsingPeer**, **chaincodeQuery**, **ledgerQuery** and **eventSource**. See how peer1 and peer2 can perform all roles as they host papercontract. Contrast to peer3, which can only be used for notifications, or ledger queries that access the blockchain component of the ledger rather than the world state, and hence do not need to have smart contracts installed. Notice how peer9 should not be used for anything other than endorsement, because those roles are better served by MagnetoCorp peers.

Again, see how the peers are described according to their logical names and their roles. Later in the profile, we’ll see the physical information for these peers.

* Line 97: organizations:

Details of all the organizations will follow, for all channels. Note that these organizations are for all channels, even though papernet is currently the only one listed. That’s because organizations can be in multiple channels, and channels can have multiple organizations. Moreover, some application operations relate to organizations rather than channels. For example, an application can request notification from one or all peers within its organization, or all organizations within the network – using [connection options](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectoptions.html). For this, there needs to be an organization to peer mapping, and this section provides it.

* Line 101: MagnetoCorp:

All peers that are considered part of MagnetoCorp are listed: peer1, peer2 and peer3. Likewise for Certificate Authorities. Again, note the logical name usages, the same as the channels: section; physical information will follow later in the profile.

* Line 121: DigiBank:

Only peer9 is listed as part of DigiBank, and no Certificate Authorities. That’s because these other peers and the DigiBank CA are not relevant for users of this connection profile.

* Line 134: orderers:

The physical information for orderers is now listed. As this connection profile only mentioned one orderer for papernet, you see orderer1.magnetocorp.example.com details listed. These include its IP address and port, and gRPC options that can override the defaults used when communicating with the orderer, if necessary. As with peers:, for high availability, specifying more than one orderer is a good idea.

* Line 152: peers:

The physical information for all previous peers is now listed. This connection profile has three peers for MagnetoCorp: peer1, peer2, and peer3; for DigiBank, a single peer peer9 has its information listed. For each peer, as with orderers, their IP address and port is listed, together with gRPC options that can override the defaults used when communicating with a particular peer, if necessary.

* Line 194: certificateAuthorities:

The physical information for certificate authorities is now listed. The connection profile has a single CA listed for MagnetoCorp, ca1-magnetocorp, and its physical information follows. As well as IP details, the registrar information allows this CA to be used for Certificate Signing Requests (CSR). These are used to request new certificates for locally generated public/private key pairs.

Now you’ve understood a connection profile for MagnetoCorp, you might like to look at a [corresponding](https://github.com/hyperledger/fabric-samples/blob/master/commercial-paper/organization/magnetocorp/gateway/networkConnection.yaml) profile for DigiBank. Locate where the profile is the same as MagnetoCorp’s, see where it’s similar, and finally where it’s different. Think about why these differences make sense for DigiBank applications.

That’s everything you need to know about connection profiles. In summary, a connection profile defines sufficient channels, organizations, peers, orderers and certificate authorities for an application to configure a gateway. The gateway allows the application to focus on business logic rather than the details of the network topology.

**Sample**

This file is reproduced inline from the GitHub commercial paper [sample](https://github.com/hyperledger/fabric-samples/blob/master/commercial-paper/organization/magnetocorp/gateway/networkConnection.yaml).

1: ---

2: #

3: # [Required]. A connection profile contains information about a set of network

4: # components. It is typically used to configure gateway, allowing applications

5: # interact with a network channel without worrying about the underlying

6: # topology. A connection profile is normally created by an administrator who

7: # understands this topology.

8: #

9: name: "papernet.magnetocorp.profile.sample"

10: #

11: # [Optional]. Analogous to HTTP, properties with an "x-" prefix are deemed

12: # "application-specific", and ignored by the gateway. For example, property

13: # "x-type" with value "hlfv1" was originally used to identify a connection

14: # profile for Fabric 1.x rather than 0.x.

15: #

16: x-type: "hlfv1"

17: #

18: # [Required]. A short description of the connection profile

19: #

20: description: "Sample connection profile for documentation topic"

21: #

22: # [Required]. Connection profile schema version. Used by the gateway to

23: # interpret these data.

24: #

25: version: "1.0"

26: #

27: # [Optional]. A logical description of each network channel; its peer and

28: # orderer names and their roles within the channel. The physical details of

29: # these components (e.g. peer IP addresses) will be specified later in the

30: # profile; we focus first on the logical, and then the physical.

31: #

32: channels:

33: #

34: # [Optional]. papernet is the only channel in this connection profile

35: #

36: papernet:

37: #

38: # [Optional]. Channel orderers for PaperNet. Details of how to connect to

39: # them is specified later, under the physical "orderers:" section

40: #

41: orderers:

42: #

43: # [Required]. Orderer logical name

44: #

45: - orderer1.magnetocorp.example.com

46: #

47: # [Optional]. Peers and their roles

48: #

49: peers:

50: #

51: # [Required]. Peer logical name

52: #

53: peer1.magnetocorp.example.com:

54: #

55: # [Optional]. Is this an endorsing peer? (It must have chaincode

56: # installed.) Default: true

57: #

58: endorsingPeer: true

59: #

60: # [Optional]. Is this peer used for query? (It must have chaincode

61: # installed.) Default: true

62: #

63: chaincodeQuery: true

64: #

65: # [Optional]. Is this peer used for non-chaincode queries? All peers

66: # support these types of queries, which include queryBlock(),

67: # queryTransaction(), etc. Default: true

68: #

69: ledgerQuery: true

70: #

71: # [Optional]. Is this peer used as an event hub? All peers can produce

72: # events. Default: true

73: #

74: eventSource: true

75: #

76: peer2.magnetocorp.example.com:

77: endorsingPeer: true

78: chaincodeQuery: true

79: ledgerQuery: true

80: eventSource: true

81: #

82: peer3.magnetocorp.example.com:

83: endorsingPeer: false

84: chaincodeQuery: false

85: ledgerQuery: true

86: eventSource: true

87: #

88: peer9.digibank.example.com:

89: endorsingPeer: true

90: chaincodeQuery: false

91: ledgerQuery: false

92: eventSource: false

93: #

94: # [Required]. List of organizations for all channels. At least one organization

95: # is required.

96: #

97: organizations:

98: #

99: # [Required]. Organizational information for MagnetoCorp

100: #

101: MagnetoCorp:

102: #

103: # [Required]. The MSPID used to identify MagnetoCorp

104: #

105: mspid: MagnetoCorpMSP

106: #

107: # [Required]. The MagnetoCorp peers

108: #

109: peers:

110: - peer1.magnetocorp.example.com

111: - peer2.magnetocorp.example.com

112: - peer3.magnetocorp.example.com

113: #

114: # [Optional]. Fabric-CA Certificate Authorities.

115: #

116: certificateAuthorities:

117: - ca-magnetocorp

118: #

119: # [Optional]. Organizational information for DigiBank

120: #

121: DigiBank:

122: #

123: # [Required]. The MSPID used to identify DigiBank

124: #

125: mspid: DigiBankMSP

126: #

127: # [Required]. The DigiBank peers

128: #

129: peers:

130: - peer9.digibank.example.com

131: #

132: # [Optional]. Orderer physical information, by orderer name

133: #

134: orderers:

135: #

136: # [Required]. Name of MagnetoCorp orderer

137: #

138: orderer1.magnetocorp.example.com:

139: #

140: # [Required]. This orderer's IP address

141: #

142: url: grpc://localhost:7050

143: #

144: # [Optional]. gRPC connection properties used for communication

145: #

146: grpcOptions:

147: ssl-target-name-override: orderer1.magnetocorp.example.com

148: #

149: # [Required]. Peer physical information, by peer name. At least one peer is

150: # required.

151: #

152: peers:

153: #

154: # [Required]. First MagetoCorp peer physical properties

155: #

156: peer1.magnetocorp.example.com:

157: #

158: # [Required]. Peer's IP address

159: #

160: url: grpc://localhost:7151

161: #

162: # [Optional]. gRPC connection properties used for communication

163: #

164: grpcOptions:

165: ssl-target-name-override: peer1.magnetocorp.example.com

166: request-timeout: 120001

167: #

168: # [Optional]. Other MagnetoCorp peers

169: #

170: peer2.magnetocorp.example.com:

171: url: grpc://localhost:7251

172: grpcOptions:

173: ssl-target-name-override: peer2.magnetocorp.example.com

174: request-timeout: 120001

175: #

176: peer3.magnetocorp.example.com:

177: url: grpc://localhost:7351

178: grpcOptions:

179: ssl-target-name-override: peer3.magnetocorp.example.com

180: request-timeout: 120001

181: #

182: # [Required]. Digibank peer physical properties

183: #

184: peer9.digibank.example.com:

185: url: grpc://localhost:7951

186: grpcOptions:

187: ssl-target-name-override: peer9.digibank.example.com

188: request-timeout: 120001

189: #

190: # [Optional]. Fabric-CA Certificate Authority physical information, by name.

191: # This information can be used to (e.g.) enroll new users. Communication is via

192: # REST, hence options relate to HTTP rather than gRPC.

193: #

194: certificateAuthorities:

195: #

196: # [Required]. MagnetoCorp CA

197: #

198: ca1-magnetocorp:

199: #

200: # [Required]. CA IP address

201: #

202: url: http://localhost:7054

203: #

204: # [Optioanl]. HTTP connection properties used for communication

205: #

206: httpOptions:

207: verify: false

208: #

209: # [Optional]. Fabric-CA supports Certificate Signing Requests (CSRs). A

210: # registrar is needed to enroll new users.

211: #

212: registrar:

213: - enrollId: admin

214: enrollSecret: adminpw

215: #

216: # [Optional]. The name of the CA.

217: #

218: caName: ca-magnetocorp

**Connection Options**

**Audience**: Architects, administrators, application and smart contract developers

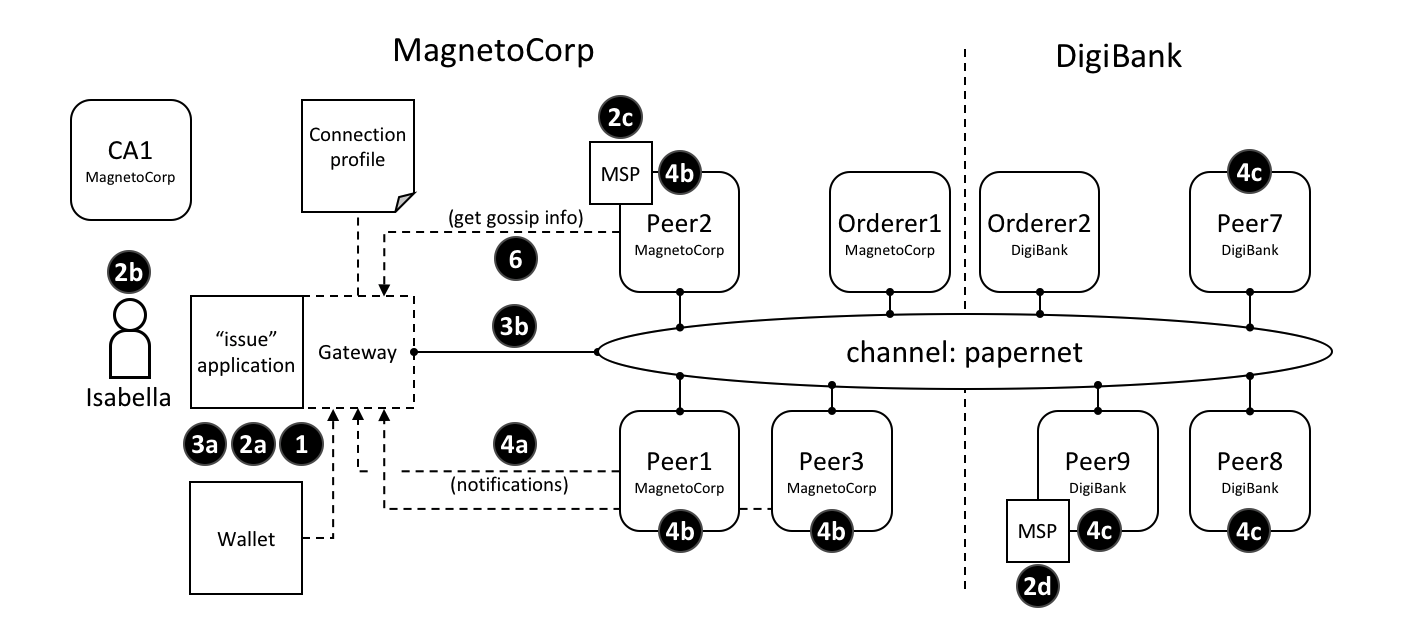
Connection options are used in conjunction with a connection profile to control *precisely* how a gateway interacts with a network. Using a gateway allows an application to focus on business logic rather than network topology.

In this topic, we’re going to cover:

* [Why connection options are important](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionoptions.html#scenario)
* [How an application uses connection options](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionoptions.html#usage)
* [What each connection option does](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionoptions.html#options)
* [When to use a particular connection option](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionoptions.html#considerations)

**Scenario**

A connection option specifies a particular aspect of a gateway’s behaviour. Gateways are important for [many reasons](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/gateway.html), the primary being to allow an application to focus on business logic and smart contracts, while it manages interactions with the many components of a network.

 *The different interaction points where connection options control behaviour. These options are explained fully in the text.*

One example of a connection option might be to specify that the gateway used by the issue application should use identity Isabella to submit transactions to the papernet network. Another might be that a gateway should wait for all three nodes from MagnetoCorp to confirm a transaction has been committed returning control. Connection options allow applications to specify the precise behaviour of a gateway’s interaction with the network. Without a gateway, applications need to do a lot more work; gateways save you time, make your application more readable, and less error prone.

**Usage**

We’ll describe the [full set](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionoptions.html#options) of connection options available to an application in a moment; let’s first see how they are specified by the sample MagnetoCorp issue application:

**const** userName **=** 'User1@org1.example.com';

**const** wallet **=** **new** FileSystemWallet('../identity/user/isabella/wallet');

**const** connectionOptions **=** {

identity**:** userName,

wallet**:** wallet,

eventHandlerOptions**:** {

commitTimeout**:** 100,

strategy**:** EventStrategies.MSPID\_SCOPE\_ANYFORTX

}

};

await gateway.connect(connectionProfile, connectionOptions);

See how the identity and wallet options are simple properties of the connectionOptions object. They have values userName and wallet respectively, which were set earlier in the code. Contrast these options with the eventHandlerOptions option which is an object in its own right. It has two properties: commitTimeout: 100 (measured in seconds) and strategy: EventStrategies.MSPID\_SCOPE\_ANYFORTX.

See how connectionOptions is passed to a gateway as a complement to connectionProfile; the network is identified by the connection profile and the options specify precisely how the gateway should interact with it. Let’s now look at the available options.

**Options**

Here’s a list of the available options and what they do.

* wallet identifies the wallet that will be used by the gateway on behalf of the application. See interaction **1**; the wallet is specified by the application, but it’s actually the gateway that retrieves identities from it.

A wallet must be specified; the most important decision is the [type](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/wallet.html#type) of wallet to use, whether that’s file system, in-memory, HSM or database.

* identity is the user identity that the application will use from wallet. See interaction **2a**; the user identity is specified by the application and represents the user of the application, Isabella, **2b**. The identity is actually retrieved by the gateway.

In our example, Isabella’s identity will be used by different MSPs (**2c**, **2d**) to identify her as being from MagnetoCorp, and having a particular role within it. These two facts will correspondingly determine her permission over resources, such as being able to read and write the ledger, for example.

A user identity must be specified. As you can see, this identity is fundamental to the idea that Hyperledger Fabric is a *permissioned* network – all actors have an identity, including applications, peers and orderers, which determines their control over resources. You can read more about this idea in the membership services [topic](https://hyperledger-fabric.readthedocs.io/en/latest/membership/membership.html).

* clientTlsIdentity is the identity that is retrieved from a wallet (**3a**) and used for secure communications (**3b**) between the gateway and different channel components, such as peers and orderers.

Note that this identity is different to the user identity. Even though clientTlsIdentity is important for secure communications, it is not as foundational as the user identity because its scope does not extend beyond secure network communications.

clientTlsIdentity is optional. You are advised to set it in production environments. You should always use a different clientTlsIdentity to identity because these identities have very different meanings and lifecycles. For example, if your clientTlsIdentity was compromised, then so would your identity; it’s more secure to keep them separate.

* eventHandlerOptions.commitTimeout is optional. It specifies, in seconds, the maximum amount of time the gateway should wait for a transaction to be committed by any peer (**4a**) before returning control to the application. The set of peers to use for notification is determined by the eventHandlerOptions.strategy option. If a commitTimeout is not specified, the gateway will use a timeout of 300 seconds.
* eventHandlerOptions.strategy is optional. It identifies the set of peers that a gateway should use to listen for notification that a transaction has been committed. For example, whether to listen for a single peer, or all peers, from its organization. It can take one of the following values:
  + EventStrategies.MSPID\_SCOPE\_ANYFORTX Listen for **any** peer within the user’s organization. In our example, see interaction points **4b**; any of peer 1, peer 2 or peer 3 from MagnetoCorp can notify the gateway.
  + EventStrategies.MSPID\_SCOPE\_ALLFORTX **This is the default value**. Listen for **all** peers within the user’s organization. In our example peer, see interaction point **4b**. All peers from MagnetoCorp must all have notified the gateway; peer 1, peer 2 and peer 3. Peers are only counted if they are known/discovered and available; peers that are stopped or have failed are not included.
  + EventStrategies.NETWORK\_SCOPE\_ANYFORTX Listen for **any** peer within the entire network channel. In our example, see interaction points **4b** and **4c**; any of peer 1-3 from MagnetoCorp or peer 7-9 of DigiBank can notify the gateway.
  + EventStrategies.NETWORK\_SCOPE\_ALLFORTX Listen for **all** peers within the entire network channel. In our example, see interaction points **4b** and **4c**. All peers from MagnetoCorp and DigiBank must notify the gateway; peers 1-3 and peers 7-9. Peers are only counted if they are known/discovered and available; peers that are stopped or have failed are not included.
  + <PluginEventHandlerFunction> The name of a user-defined event handler. This allows a user to define their own logic for event handling. See how to [define](https://hyperledger.github.io/fabric-sdk-node/master/tutorial-transaction-commit-events.html) a plugin event handler, and examine a [sample handler](https://github.com/hyperledger/fabric-sdk-node/blob/master/test/integration/network-e2e/sample-transaction-event-handler.js).

A user-defined event handler is only necessary if you have very specific event handling requirements; in general, one of the built-in event strategies will be sufficient. An example of a user-defined event handler might be to wait for more than half the peers in an organization to confirm a transaction has been committed.

If you do specify a user-defined event handler, it does not affect your application logic; it is quite separate from it. The handler is called by the SDK during processing; it decides when to call it, and uses its results to select which peers to use for event notification. The application receives control when the SDK has finished its processing.

If a user-defined event handler is not specified then the default values for EventStrategies are used.

* discovery.enabled is optional and has possible values true or false. The default is true. It determines whether the gateway uses [service discovery](https://hyperledger-fabric.readthedocs.io/en/latest/discovery-overview.html) to augment the network topology specified in the connection profile. See interaction point **6**; peer’s gossip information used by the gateway.

This value will be overridden by the INITIALIIZE-WITH-DISCOVERY environment variable, which can be set to true or false.

* discovery.asLocalhost is optional and has possible values true or false. The default is true. It determines whether IP addresses found during service discovery are translated from the docker network to the local host.

Typically developers will write applications that use docker containers for their network components such as peers, orderers and CAs, but that do not run in docker containers themselves. This is why true is the default; in production environments, applications will likely run in docker containers in the same manner as network components and therefore address translation is not required. In this case, applications should either explicitly specify false or use the environment variable override.

This value will be overridden by the DISCOVERY-AS-LOCALHOST environment variable, which can be set to true or false.

**Considerations**

The following list of considerations is helpful when deciding how to choose connection options.

* eventHandlerOptions.commitTimeout and eventHandlerOptions.strategy work together. For example, commitTimeout: 100 and strategy: EventStrategies.MSPID\_SCOPE\_ANYFORTX means that the gateway will wait for up to 100 seconds for *any* peer to confirm a transaction has been committed. In contrast, specifying strategy: EventStrategies.NETWORK\_SCOPE\_ALLFORTX means that the gateway will wait up to 100 seconds for *all* peers in *all* organizations.
* The default value of eventHandlerOptions.strategy: EventStrategies.MSPID\_SCOPE\_ALLFORTX will wait for all peers in the application’s organization to commit the transaction. This is a good default because applications can be sure that all their peers have an up-to-date copy of the ledger, minimizing concurrency issues

However, as the number of peers in an organization grows, it becomes a little unnecessary to wait for all peers, in which case using a pluggable event handler can provide a more efficient strategy. For example the same set of peers could be used to submit transactions and listen for notifications, on the safe assumption that consensus will keep all ledgers synchronized.

* Service discovery requires clientTlsIdentity to be set. That’s because the peers exchanging information with an application need to be confident that they are exchanging information with entities they trust. If clientTlsIdentity is not set, then discovery will not be obeyed, regardless of whether or not it is set.
* Although applications can set connection options when they connect to the gateway, it can be necessary for these options to be overridden by an administrator. That’s because options relate to network interactions, which can vary over time. For example, an administrator trying to understand the effect of using service discovery on network performance.

A good approach is to define application overrides in a configuration file which is read by the application when it configures its connection to the gateway.

Because the discovery options enabled and asLocalHost are most frequently required to be overridden by administrators, the environment variables INITIALIIZE-WITH-DISCOVERY and DISCOVERY-AS-LOCALHOST are provided for convenience. The administrator should set these in the production runtime environment of the application, which will most likely be a docker container.

# Wallet

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

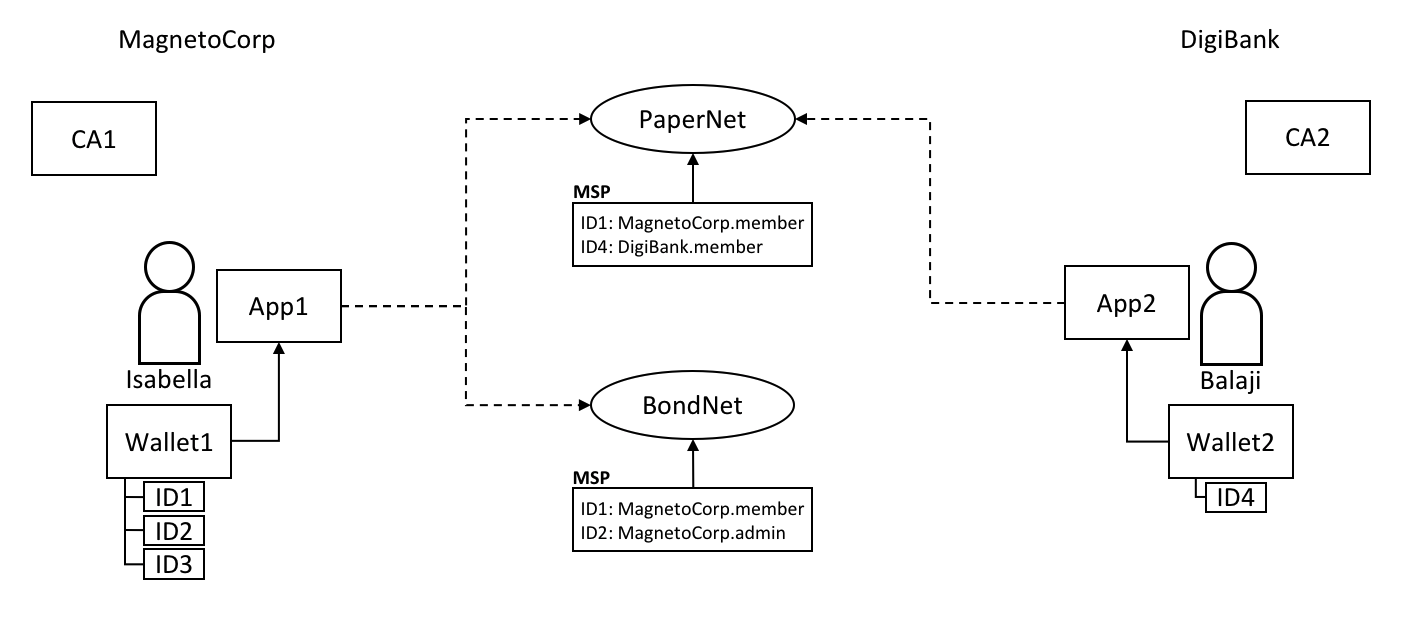
A wallet contains a set of user identities. An application run by a user selects one of these identities when it connects to a channel. Access rights to channel resources, such as the ledger, are determined using this identity in combination with an MSP.

In this topic, we’re going to cover:

* [Why wallets are important](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/wallet.html#scenario)
* [How wallets are organized](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/wallet.html#structure)
* [Different types of wallet](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/wallet.html#types)
* [Wallet operations](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/wallet.html#operations)

## Scenario

When an application connects to a network channel such as PaperNet, it selects a user identity to do so, for example ID1. The channel MSPs associate ID1 with a role within a particular organization, and this role will ultimately determine the application’s rights over channel resources. For example, ID1 might identify a user as a member of the MagnetoCorp organization who can read and write to the ledger, whereas ID2 might identify an administrator in MagnetoCorp who can add a new organization to a consortium.

 Two users, Isabella and Balaji have wallets containing different identities they can use to connect to different network channels, PaperNet and BondNet.

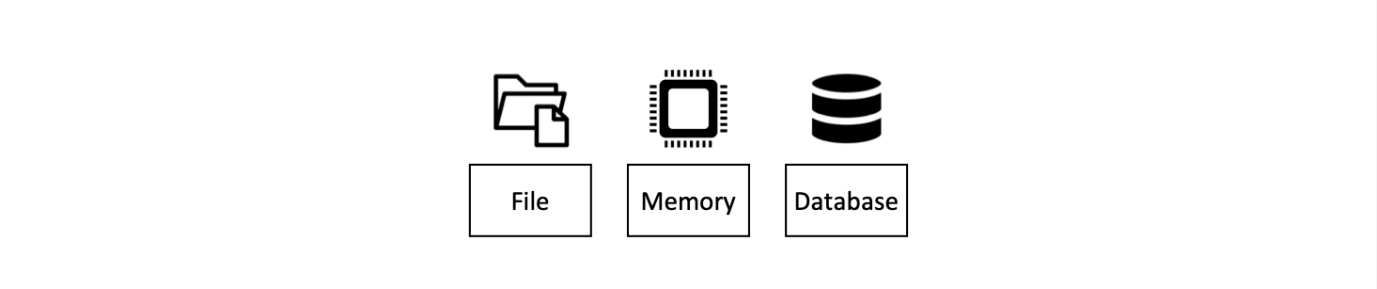
Consider the example of two users; Isabella from MagnetoCorp and Balaji from DigiBank. Isabella is going to use App 1 to invoke a smart contract in PaperNet and a different smart contract in BondNet. Similarly, Balaji is going to use App 2 to invoke smart contracts, but only in PaperNet. (It’s very [easy](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/application.html#construct-request) for applications to access multiple networks and multiple smart contracts within them.)

See how:

* MagnetoCorp uses CA1 to issue identities and DigiBank uses CA2 to issue identities. These identities are stored in user wallets.
* Balaji’s wallet holds a single identity, ID4 issued by CA2. Isabella’s wallet has many identities, ID1, ID2 and ID3, issued by CA1. Wallets can hold multiple identities for a single user, and each identity can be issued by a different CA.
* Both Isabella and Balaji connect to PaperNet, and its MSPs determine that Isabella is a member of the MagnetoCorp organization, and Balaji is a member of the DigiBank organization, because of the respective CAs that issued their identities. (It is [possible](https://hyperledger-fabric.readthedocs.io/en/latest/membership/membership.html#mapping-msps-to-organizations) for an organization to use multiple CAs, and for a single CA to support multiple organizations.)
* Isabella can use ID1 to connect to both PaperNet and BondNet. In both cases, when Isabella uses this identity, she is recognized as a member of MangetoCorp.
* Isabella can use ID2 to connect to BondNet, in which case she is identified as an administrator of MagnetoCorp. This gives Isabella two very different privileges: ID1 identifies her as a simple member of MagnetoCorp who can read and write to the BondNet ledger, whereas ID2 identities her as a MagnetoCorp administrator who can add a new organization to BondNet.
* Balaji cannot connect to BondNet with ID4. If he tried to connect, ID4 would not be recognized as belonging to DigiBank because CA2 is not known to BondNet’s MSP.

## Types

There are different types of wallets according to where they store their identities:

 The three different types of wallet storage: File system, In-memory and CouchDB.

* **File system**: This is the most common place to store wallets; file systems are pervasive, easy to understand, and can be network mounted. They are a good default choice for wallets.
* **In-memory**: A wallet in application storage. Use this type of wallet when your application is running in a constrained environment without access to a file system; typically a web browser. It’s worth remembering that this type of wallet is volatile; identities will be lost after the application ends normally or crashes.
* **CouchDB**: A wallet stored in CouchDB. This is the rarest form of wallet storage, but for those users who want to use the database back-up and restore mechanisms, CouchDB wallets can provide a useful option to simplify disaster recovery.

Use factory functions provided by the Wallets [class](https://hyperledger.github.io/fabric-sdk-node/master/module-fabric-network.Wallets.html) to create wallets.

### Hardware Security Module

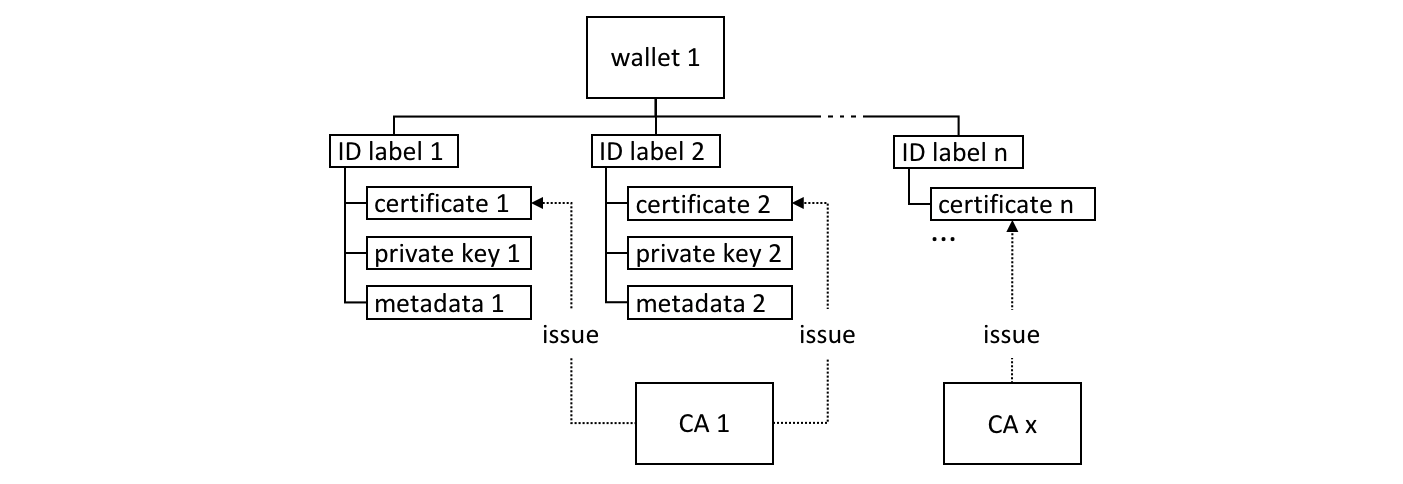
A Hardware Security Module (HSM) is an ultra-secure, tamper-proof device that stores digital identity information, particularly private keys. HSMs can be locally attached to your computer or network accessible. Most HSMs provide the ability to perform on-board encryption with private keys, such that the private keys never leave the HSM.

An HSM can be used with any of the wallet types. In this case the certificate for an identity will be stored in the wallet and the private key will be stored in the HSM.

To enable the use of HSM-managed identities, an IdentityProvider must be configured with the HSM connection information and registered with the wallet. For further details, refer to the [Using wallets to manage identities](https://hyperledger.github.io/fabric-sdk-node/master/tutorial-wallet.html) tutorial.

## Structure

A single wallet can hold multiple identities, each issued by a particular Certificate Authority. Each identity has a standard structure comprising a descriptive label, an X.509 certificate containing a public key, a private key, and some Fabric-specific metadata. Different [wallet types](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/wallet.html#types) map this structure appropriately to their storage mechanism.

 A Fabric wallet can hold multiple identities with certificates issued by a different Certificate Authority. Identities comprise certificate, private key and Fabric metadata.

There’s a couple of key class methods that make it easy to manage wallets and identities:

**const** identity**:** X509Identity **=** {

credentials**:** {

certificate**:** certificatePEM,

privateKey**:** privateKeyPEM,

},

mspId**:** 'Org1MSP',

type**:** 'X.509',

};

await wallet.put(identityLabel, identity);

See how an identity is created that has metadata Org1MSP, a certificate and a privateKey. See how wallet.put() adds this identity to the wallet with a particular identityLabel.

The Gateway class only requires the mspId and type metadata to be set for an identity – Org1MSP and X.509 in the above example. It currently uses the MSP ID value to identify particular peers from a [connection profile](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionprofile.html), for example when a specific notification [strategy](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectoptions.html) is requested. In the DigiBank gateway file networkConnection.yaml, see how Org1MSP notifications will be associated with peer0.org1.example.com:

organizations:

Org1:

mspid: Org1MSP

peers:

- peer0.org1.example.com

You really don’t need to worry about the internal structure of the different wallet types, but if you’re interested, navigate to a user identity folder in the commercial paper sample:

magnetocorp**/**identity**/**user**/**isabella**/**

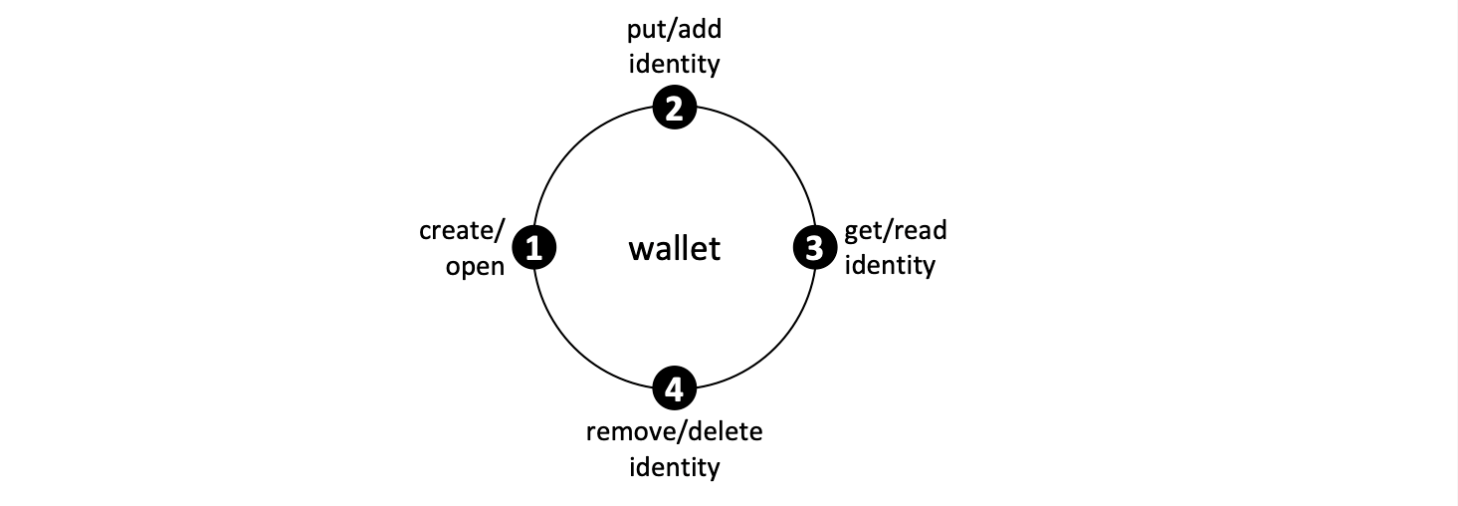
wallet**/**

User1@org1**.**example**.**com**.**id

You can examine these files, but as discussed, it’s easier to use the SDK to manipulate these data.

## Operations

The different wallet types all implement a common [Wallet](https://hyperledger.github.io/fabric-sdk-node/master/module-fabric-network.Wallet.html) interface which provides a standard set of APIs to manage identities. It means that applications can be made independent of the underlying wallet storage mechanism; for example, File system and HSM wallets are handled in a very similar way.

 Wallets follow a lifecycle: they can be created or opened, and identities can be read, added and deleted.

An application can use a wallet according to a simple lifecycle. Wallets can be opened or created, and subsequently identities can be added, updated, read and deleted. Spend a little time on the different Wallet methods in the [JSDoc](https://hyperledger.github.io/fabric-sdk-node/master/module-fabric-network.Wallet.html) to see how they work; the commercial paper tutorial provides a nice example in addToWallet.js:

**const** wallet **=** await Wallets.newFileSystemWallet('../identity/user/isabella/wallet');

**const** cert **=** fs.readFileSync(path.join(credPath, '.../User1@org1.example.com-cert.pem')).toString();

**const** key **=** fs.readFileSync(path.join(credPath, '.../\_sk')).toString();

**const** identityLabel **=** 'User1@org1.example.com';

**const** identity **=** {

credentials**:** {

certificate**:** cert,

privateKey**:** key,

},

mspId**:** 'Org1MSP',

type**:** 'X.509',

};

await wallet.put(identityLabel, identity);

Notice how:

* When the program is first run, a wallet is created on the local file system at .../isabella/wallet.
* a certificate cert and private key are loaded from the file system.
* a new X.509 identity is created with cert, key and Org1MSP.
* the new identity is added to the wallet with wallet.put() with a label User1@org1.example.com.

That’s everything you need to know about wallets. You’ve seen how they hold identities that are used by applications on behalf of users to access Fabric network resources. There are different types of wallets available depending on your application and security needs, and a simple set of APIs to help applications manage wallets and the identities within them.

**Gateway**

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

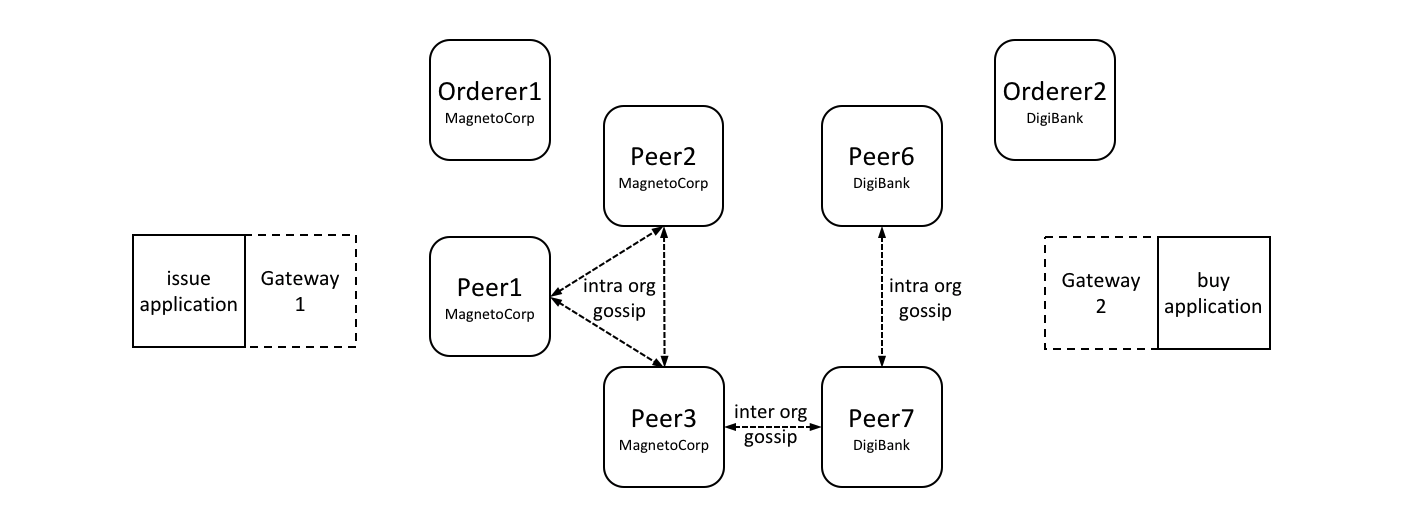
A gateway manages the network interactions on behalf of an application, allowing it to focus on business logic. Applications connect to a gateway and then all subsequent interactions are managed using that gateway’s configuration.

In this topic, we’re going to cover:

* [Why gateways are important](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/gateway.html#scenario)
* [How applications use a gateway](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/gateway.html#connect)
* [How to define a static gateway](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/gateway.html#static)
* [How to define a dynamic gateway for service discovery](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/gateway.html#dynamic)
* [Using multiple gateways](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/gateway.html#multiple-gateways)

**Scenario**

A Hyperledger Fabric network channel can constantly change. The peer, orderer and CA components, contributed by the different organizations in the network, will come and go. Reasons for this include increased or reduced business demand, and both planned and unplanned outages. A gateway relieves an application of this burden, allowing it to focus on the business problem it is trying to solve.

 *A MagnetoCorp and DigiBank applications (issue and buy) delegate their respective network interactions to their gateways. Each gateway understands the network channel topology comprising the multiple peers and orderers of two organizations MagnetoCorp and DigiBank, leaving applications to focus on business logic. Peers can talk to each other both within and across organizations using the gossip protocol.*

A gateway can be used by an application in two different ways:

* **Static**: The gateway configuration is *completely* defined in a [connection profile](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionprofile.html). All the peers, orderers and CAs available to an application are statically defined in the connection profile used to configure the gateway. For peers, this includes their role as an endorsing peer or event notification hub, for example. You can read more about these roles in the connection profile [topic](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionprofile.html).

The SDK will use this static topology, in conjunction with gateway [connection options](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionoptions), to manage the transaction submission and notification processes. The connection profile must contain enough of the network topology to allow a gateway to interact with the network on behalf of the application; this includes the network channels, organizations, orderers, peers and their roles.

* **Dynamic**: The gateway configuration is minimally defined in a connection profile. Typically, one or two peers from the application’s organization are specified, and they use [service discovery](https://hyperledger-fabric.readthedocs.io/en/latest/discovery-overview.html) to discover the available network topology. This includes peers, orderers, channels, deployed smart contracts and their endorsement policies. (In production environments, a gateway configuration should specify at least two peers for availability.)

The SDK will use all of the static and discovered topology information, in conjunction with gateway connection options, to manage the transaction submission and notification processes. As part of this, it will also intelligently use the discovered topology; for example, it will *calculate* the minimum required endorsing peers using the discovered endorsement policy for the smart contract.

You might ask yourself whether a static or dynamic gateway is better? The trade-off is between predictability and responsiveness. Static networks will always behave the same way, as they perceive the network as unchanging. In this sense they are predictable – they will always use the same peers and orderers if they are available. Dynamic networks are more responsive as they understand how the network changes – they can use newly added peers and orderers, which brings extra resilience and scalability, at potentially some cost in predictability. In general it’s fine to use dynamic networks, and indeed this the default mode for gateways.

Note that the *same* connection profile can be used statically or dynamically. Clearly, if a profile is going to be used statically, it needs to be comprehensive, whereas dynamic usage requires only sparse population.

Both styles of gateway are transparent to the application; the application program design does not change whether static or dynamic gateways are used. This also means that some applications may use service discovery, while others may not. In general using dynamic discovery means less definition and more intelligence by the SDK; it is the default.

**Connect**

When an application connects to a gateway, two options are provided. These are used in subsequent SDK processing:

await gateway.connect(connectionProfile, connectionOptions);

* **Connection profile**: connectionProfile is the gateway configuration that will be used for transaction processing by the SDK, whether statically or dynamically. It can be specified in YAML or JSON, though it must be converted to a JSON object when passed to the gateway:
* **let** connectionProfile **=** yaml.safeLoad(fs.readFileSync('../gateway/paperNet.yaml', 'utf8'));

Read more about [connection profiles](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionprofile.html) and how to configure them.

* **Connection options**: connectionOptions allow an application to declare rather than implement desired transaction processing behaviour. Connection options are interpreted by the SDK to control interaction patterns with network components, for example to select which identity to connect with, or which peers to use for event notifications. These options significantly reduce application complexity without compromising functionality. This is possible because the SDK has implemented much of the low level logic that would otherwise be required by applications; connection options control this logic flow.

Read about the list of available [connection options](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/connectionoptions.html) and when to use them.

**Static**

Static gateways define a fixed view of a network. In the MagnetoCorp [scenario](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/gateway.html#scenario), a gateway might identify a single peer from MagnetoCorp, a single peer from DigiBank, and a MagentoCorp orderer. Alternatively, a gateway might define *all* peers and orderers from MagnetCorp and DigiBank. In both cases, a gateway must define a view of the network sufficient to get commercial paper transactions endorsed and distributed.

Applications can use a gateway statically by explicitly specifying the connect option discovery: { enabled:false } on the gateway.connect() API. Alternatively, the environment variable setting FABRIC\_SDK\_DISCOVERY=false will always override the application choice.

Examine the [connection profile](https://github.com/hyperledger/fabric-samples/blob/master/commercial-paper/organization/magnetocorp/gateway/networkConnection.yaml) used by the MagnetoCorp issue application. See how all the peers, orderers and even CAs are specified in this file, including their roles.

It’s worth bearing in mind that a static gateway represents a view of a network at a *moment in time*. As networks change, it may be important to reflect this in a change to the gateway file. Applications will automatically pick up these changes when they re-load the gateway file.

**Dynamic**

Dynamic gateways define a small, fixed *starting point* for a network. In the MagnetoCorp [scenario](https://hyperledger-fabric.readthedocs.io/en/latest/developapps/gateway.html#scenario), a dynamic gateway might identify just a single peer from MagnetoCorp; everything else will be discovered! (To provide resiliency, it might be better to define two such bootstrap peers.)

If [service discovery](https://hyperledger-fabric.readthedocs.io/en/latest/discovery-overview.html) is selected by an application, the topology defined in the gateway file is augmented with that produced by this process. Service discovery starts with the gateway definition, and finds all the connected peers and orderers within the MagnetoCorp organization using the [gossip protocol](https://hyperledger-fabric.readthedocs.io/en/latest/gossip.html). If [anchor peers](https://hyperledger-fabric.readthedocs.io/en/latest/glossary.html#anchor-peer) have been defined for a channel, then service discovery will use the gossip protocol across organizations to discover components within the connected organization. This process will also discover smart contracts installed on peers and their endorsement policies defined at a channel level. As with static gateways, the discovered network must be sufficient to get commercial paper transactions endorsed and distributed.

Dynamic gateways are the default setting for Fabric applications. They can be explicitly specified using the connect option discovery: { enabled:true } on the gateway.connect() API. Alternatively, the environment variable setting FABRIC\_SDK\_DISCOVERY=true will always override the application choice.

A dynamic gateway represents an up-to-date view of a network. As networks change, service discovery will ensure that the network view is an accurate reflection of the topology visible to the application. Applications will automatically pick up these changes; they do not even need to re-load the gateway file.

**Multiple gateways**

Finally, it is straightforward for an application to define multiple gateways, both for the same or different networks. Moreover, applications can use the name gateway both statically and dynamically.

It can be helpful to have multiple gateways. Here are a few reasons:

* Handling requests on behalf of different users.
* Connecting to different networks simultaneously.
* Testing a network configuration, by simultaneously comparing its behaviour with an existing configuration.