Developing DApps on Oracle Blockchain Platform

A Developer's Guide

developer.oracle.com

ORACLE

Developing DApps on Oracle Blockchain Platform

Many organizations are adopting blockchain technology to increase the speed of business-to-business transactions and share data in a secure and tamper-proof system. Blockchain, the ingenious technology behind cryptocurrencies, has very broad applications in areas such as ERP, supply chain, healthcare, and financial services. The concept at the core of blockchain is a distributed ledger consisting of a current set of facts maintained in a world state key-value database and transaction history. This is maintained as a list of blocks, with the database changes and transaction history extended by consensus via peer-to-peer protocols, and linked and secured with cryptographic hashes.

Blockchain's use of distributed ledger technology, multi-party consensus, and embedded cryptography for signing transactions and linking blocks, enables a single source of truth among multiple participants with data integrity, high availability and durability of data, resilience against single point of failure and malicious attacks, and process integrity. These attributes can be leveraged to optimize an ecosystem of participants via faster transactions with near real-time settlement, lower costs by avoiding intermediaries, and greater automation of verification, settlement, and downstream event triggers.

In order to ensure blockchain can deliver these capabilities and support multiple participating organizations relying upon it, the underlying framework must have highly credible architecture. The Oracle Blockchain Platform (OBP) builds on Linux Foundation's Hyperledger Fabric open-source code, a global, cross-industry collaborative effort to bring blockchain technology for business into the mainstream.

This community with representatives from multiple technology companies and many industries ensures the transparency, longevity, and interoperability necessary for growing blockchain adoption in enterprise and government deployments around the world.

The benefits of blockchain are faster, more automated processes, which is a distinct benefit to developers. Instead of building the necessary infrastructure on premises, get an accelerated start with Oracle's comprehensive, enterprise-grade Blockchain-as-a-Service offering, that adds a number of unique advantages to the open source foundation. With the Oracle Blockchain Platform providing a complete, pre-assembled blockchain platform with a comprehensive set of tools and APIs, developers are able to quickly get a blockchain network up and running.

For development and testing, a free downloadable OBP SDK is also available with a time-limited license.

Free SDK →

Free Cloud Trial →

This allows developers to focus resources on building and integrating the application and its underlying smart contract code specific to their blockchain network, rather than spending a lot of time and effort on building and hardening the infrastructure of the blockchain platform. The hardened, industrialized platform, its dynamic configuration and monitoring tools, and a powerful set of integrated enterprise features helps to reduce the time to develop and integrate applications, resulting in higher productivity and significantly reduced operating costs. Developers can deliver faster results for rapid experimentation and leverage production-strength platform for mission-critical deployment of a trusted blockchain network that provides increased visibility, faster settlement, and decreased risk of fraud for business-to-business transactions.

Contents

Free SDK →

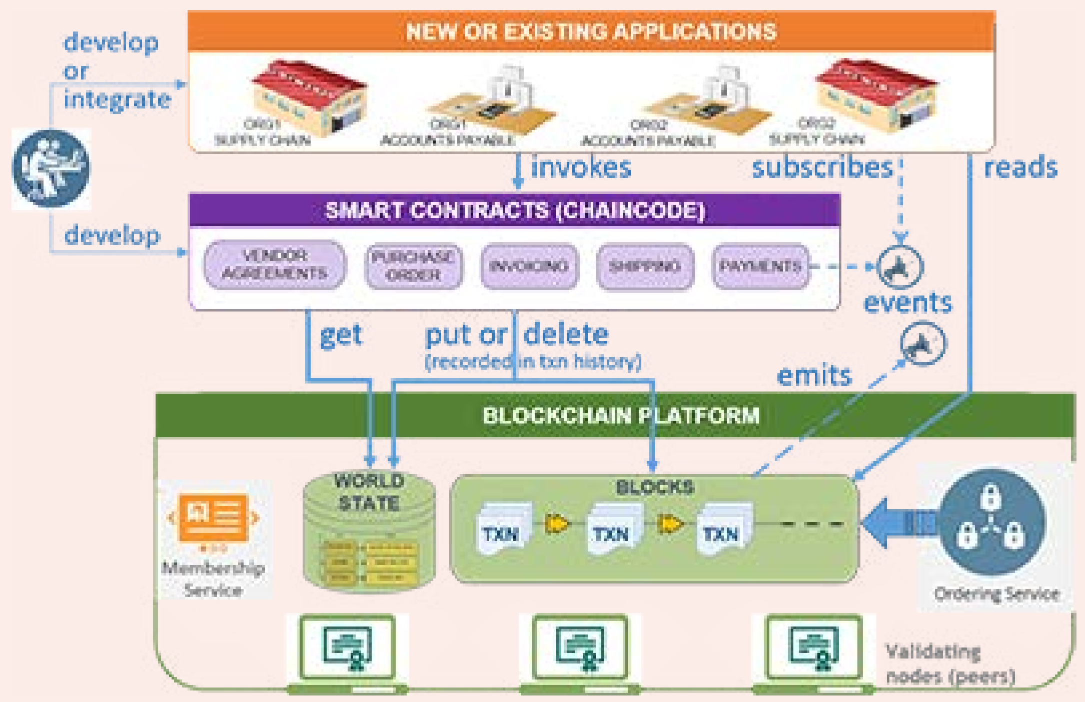
Free Cloud Trial →

What is Oracle Blockchain Platform?

A production-ready permissioned blockchain network consists of validating nodes, ordering service, and membership services for enrolling organizations, supported by a broad set of dependencies, including compute, storage, event management, and identity management. It needs to provide the interfaces to maintain ledgers, create channels, add participating organizations and nodes, invoke and monitor smart contracts, browse the ledge, and have the APIs necessary to build and run your applications.

Oracle Blockchain Platform (OBP) provides this comprehensive set of nodes and services along with all the required dependencies and APIs, architected to support highly scalable distributed transaction processing required by enterprise applications.

Blockchain can be very resource (compute, memory, storage and network) intensive, so performance and scalability are critical characteristics of a successful service. Blockchain system can be visualized in three layers as illustrated in the diagram below:



Blockchain Platform includes

- A network of validating nodes (peers)
- Distributed ledger (linked blocks, world state and history DB)
- Ordering service (for creating blocks)
- Membership services (for managing organizations in a permissioned blockchain)

Smart Contracts (Chaincode) layer consists of chaincode programs that contain the business logic for updating the ledger, querying data, and/or publishing events. Chaincodes can read the ledger data to verify conditions as part of any proposed updates or deletes and trigger custom events. Note that updates and deletes are proposed (or simulated) and are not final until transactions are committed following consensus and validation protocols.

New or Existing Applications, which can

- Register/enroll organizations as members
- Submit transactions (invoke smart contracts) to update or query data
- Consume events emitted by the chaincodes or by the blockchain platform

Oracle Blockchain Platform (OBP) provides the following capabilities for the lowest layer in this diagram:

- An ability to spin up one or more pre-assembled instances of a blockchain platform cloud service in Oracle Cloud Infrastructure (OCI)
- A private cloud service deployable as software appliance outside of OCI
- An ability to link instances into hybrid blockchain networks, spanning Oracle and non-Oracle Hyperledger Fabric nodes
- An administrative console and set of tools for administration and monitoring of OBP instances
- Tools to deploy and manage smart contracts
- APIs and SDKs for applications to interact with chaincode and consume events

The creation of smart contracts and integration with the applications that invoke them is left to customers or system integrators.

Note that some Oracle ISV partners provide complete solutions based on OBP, which include smart contracts, new application components, and integrations with Oracle and 3rd party applications (see some of their solutions on Oracle Cloud Marketplace.)

As a developer, once you have registered an account on Oracle Cloud (you can start at cloud.oracle.com/blockchain), you can setup a new instance of OBP and start developing your chaincode and integrate applications within minutes. You can easily integrate new or existing applications using the OBP REST API or broad portfolio of enterprise integration adapters from Oracle Integration Cloud (OIC.) Enterprise developers can securely extend existing business processes with real-time data sharing across existing Oracle ERP Cloud, SCM Cloud, HCM Cloud, CX Cloud, Netsuite Suite Cloud Platform, Flexcube core banking, and custom applications on premise or in a cloud.

Understanding Blockchain Networks, Nodes, and other Components

The terms defined below are specific to Hyperledger Fabric blockchain architecture on which Oracle Blockchain Platform is based.

Blockchain Network: Collection of member organizations that can read and write to the blockchain ledger. Organizations can be founders, which run the ordering service, or participants, which join their peer nodes to founder's ordering service.

Nodes: Distinct operating entities on the network. Nodes include the orderer(s), certificate authority, REST proxy, and peers.

Orderers: Node for creating new blocks from the transactions sent by the clients. Orderer can operate by itself (Solo) or as part of an ordering cluster, where they are also referred to as ordering service nodes or OSNs.

Peers: Participation validates notes of member organizations. Peers are responsible for maintaining a copy of the ledger, running smart contracts, and committing transactions. Organizations can have one or more peers on the network. Two peers per member is a common configuration for operating performance and availability.

Channel: A subnet within a blockchain network with an isolated ledger and an authorized group of member organizations represented by their peer nodes. Peers executing a transaction targeting a specific channel can only access the ledger on that channel, but a single peer can belong to multiple channels and execute independent transactions on each of them.

Chaincode: a synonym for smart contract, the specific programming module that contains business logic invoked on the channel to read and write application data from/to the ledger, apply any validation logic, and trigger application events.

Chaincodes are installed on and specific to a channel and its peers. The same chaincode can be instantiated and used on multiple channels, however, while executing a transaction request it operates in the context of the specific channel and its ledger.

Certificate Authority: a node providing PKI functionality in the blockchain. CA is responsible for registering member organizations, issuing their enrollment and TLS certificates, and handling certificate revocation and renewal.

OBP is pre-assembled with all these components and their underlying dependencies required for out-of-the-box operation: event management (Kafka), identity management (IDCS), object store (OSS), two manager VMs in high availability configuration for hosting all control nodes, agents providing embedded backup and recovery and telemetry for cloud monitoring and management operations, and a separate VM to isolate chaincode execution containers running customer code.

The pre-assembled nature of OBP reduces the burden of building an open source environment from scratch and provides a blockchain platform you can provision with a simple request and a few clicks in minutes using QuickStart templates sized for development and production needs.

The integration of Oracle Identity Cloud Service in the OBP CS is a unique differentiator, which provides strong identity management with identity federation for authentication and adding new members, protects secure access with behavioral authentication and single sign-on. For an overview of OBP Cloud Service please review the "Getting Started with Oracle Blockchain Cloud Service" video.

Free SDK →

Free Cloud Trial →

Transaction Flow

Oracle leverages Hyperledger Fabric, a Linux Foundation open source project, as the foundation of its blockchain platform. The transaction flow is defined by Fabric protocols between peers, orderers, fabric-ca, and clients as detailed on page 8.

Fabric transactions fall into two categories: invocations (usually updates) and queries. Both invoke smart contracts and request an endorsement – a digitally signed result of the execution. In addition, invocations add a further stage – a commit, where the transaction is included in a block and, after verification, appended to the ledger by all peers on the channel specified at the invocation time. Note that commit stage is at client's discretion, and can be optionally added even for queries or failed invocations if it's important to record them in the ledger.

Endorsement

Endorsement is the process of executing chaincode and returning results to a client. The client request for particular chaincode invocation is sent to a number of peers (as defined by chaincodes endorsement policy) on a specific channel. The response includes the return code and Read-Write set (RWset), which includes all keys read and updated during the execution, signed by the peer's private key, with the signature as transaction ID that provides proof of execution. Peers do not write the results into the ledger in this phase. If chaincode endorsement policy specifies multiple endorsements, the execution is performed on the multiple peers, and the client receives multiple responses. It is the client's responsible for comparing the results to ensure they match, and checking that appropriate number of responses has been received consistent with the endorsement policy.

Commit

Commit is the process of validating and writing transactions to the ledger. During this step, the client sends the response and all the peer signatures to the orderers. Upon validating the signatures, ordering service (a cluster of orderers) sequences transactions from all the clients and group them in blocks, which are then delivered to the peers on a relevant channel. Each peer then validates each transaction in the block by checking for sufficient endorsements and verifying that the relevant data in the ledger have not been changed by another transaction since the current one was executed in the endorsement stage (i.e., if at endorsement time Alice has a balance of $100 and transaction is to transfer $70 to Bob, we need to know that there haven't been intervening transactions that might have reduced Alice's balance before we commit the $70 transfer.) Once these checks are complete, each peer writes the valid transactions to the world state database, updates the history database, and returns transaction commit event.

Note that the block is added to the chain with transaction flag indicating commit status for each transaction. If verification step has failed, the transaction will be marked as failed in the block and failure notification returned to the client, which can re-submit it if appropriate.

Query

Query is the process of reading the current ledger state through chaincode query functions and returning an endorsed result payload. Queries do not typically change the ledger state, so are not normally sent for commitment. However, client applications can submit queries to be committed as an auditable proof that a peer had knowledge of the ledger state at a specific point in time.

In summary, the end-to-end transaction flow as depicted in the diagram on page 9:

1. The client applications register and enroll with fabric-ca to obtain a certificate.

2. Client sends a signed transaction proposal request to endorsing peer nodes (one or more as specified in the endorsement policy).

3. Endorsing peers verify the signature and execute the specified chaincode on the specified channel.

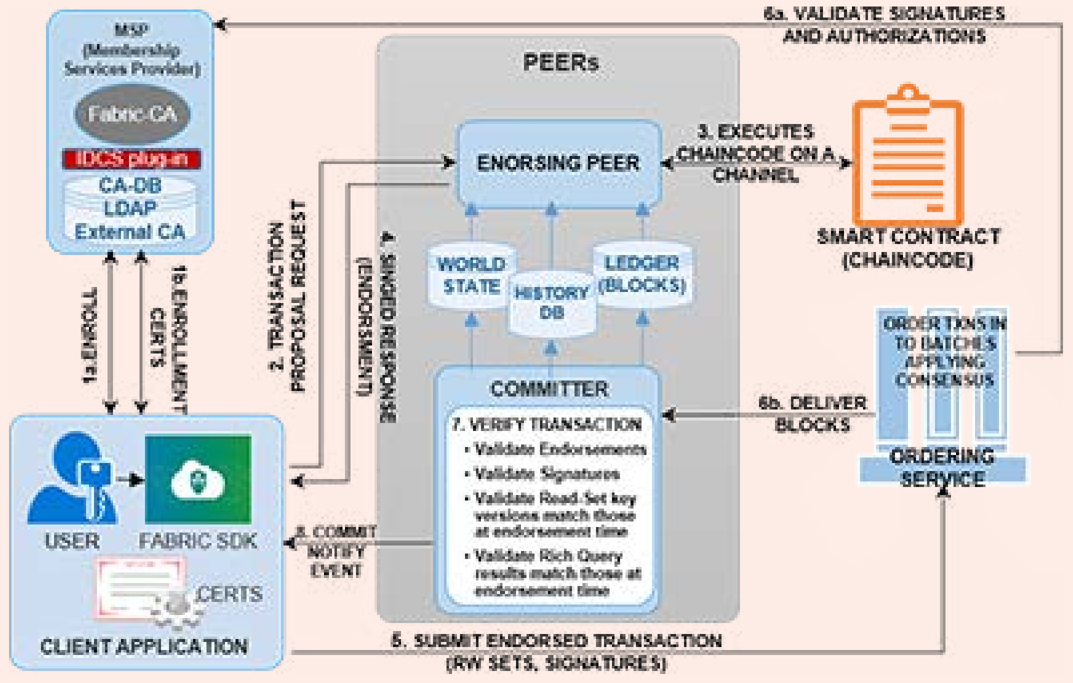
4. Chaincode responses (RWSet) are signed by the peers and these endorsements are returned to the client.

5. If responses are valid, results from the peers match, and sufficient number has been received as per the chaincode endorsement policy requirements, the client assembles the endorsements into a commit package and sends it to ordering service to be included in a block.

6. Orderers validate the signatures and sequence transactions into blocks, sending the new blocks to the peers for the specific channel.

7. Committers in each peer validate the transaction signatures and check if there's a sufficient number of signatures to meet endorsement policy. They also verify that the values of the keys read at endorsement time haven't changed since endorsement time. And in OBCS they also verify that rich query results haven't changed. If everything's valid, the transaction is committed, i.e., updated key values are stored into the world state database, the commit flag for the transaction is set in the block added to the chain, and history database is updated with appropriate pointers. Note that blocks are appended to the chain irrespective of whether transactions are valid or not – a commit flag for each transaction is used to indicate if it has committed or failed.

8. The peer emits transaction commit and new block added events, as well as any custom events declared in the chaincode logic and these are delivered to any subscribing client.


MSP (Membership Services Provider)
Fabric-CA
IDCS plug-in
CA-DB
LDAP
External CA
1a.ENROLL
1b.ENROLLMENT CERTS
2. TRANSACTION PROPOSAL REQUEST
4. SIGNED RESPONSE (ENDORSMENT)
PEERs
ENORSING PEER
3. EXECUTES CHAINCODE ON A CHANNEL
SMART CONTRACT (CHAINCODE)
WORLD STATE
HISTORY DB
LEDGER (BLOCKS)
COMMITTER
7. VERIFY TRANSACTION
• Validate Endorsements
• Validate Signatures
• Validate Read-Set key versions match those at endorsement time
• Validate Rich Query results match those at endorsement time
6a. VALIDATE SIGNATURES AND AUTHORIZATIONS
ORDER TXNS IN TO BATCHES APPLYING CONSENSUS
6b. DELIVER BLOCKS
ORDERING SERVICE
USER
FABRIC SDK
CERTS
CLIENT APPLICATION
8. COMMIT NOTIFY EVENT
5. SUBMIT ENDORSED TRANSACTION (RW SETS, SIGNATURES)

In addition to the main transaction flow protocol, the Fabric blockchain peers use a messaging protocol called gossip data dissemination protocol to continuously send messages to each other and keep the ledger copies current.

Gossip messages will identify available peers, detect ones that are offline, update blocks to peers that are out of sync, and quickly bring new peers' ledger copy up to the latest state.

This architecture imposes significant responsibilities on a client to orchestrate the transaction flow, and handle async events using an SDK. In order to simplify working with the blockchain, OBP provides a REST proxy (a.k.a. Gateway) that handles the orchestration and exposes RESTful endpoints for applications to trigger transactions, query the ledger, and subscribe to events. See Invoking Chaincodes via REST API below for more details.

In addition to the REST API for interacting with transactions and events, OBCS also provides an extensive REST API for configuration and monitoring of the blockchain network. These administrative functions can be integrated into any DevOps toolset, and enable deployment of chaincode, adding nodes, creating channels, adding organizations, querying and setting configuration attributes, starting/stopping components and many other tasks.

Provisioning a Blockchain Network

The key properties of a permissioned blockchain implementation are that all members of the network see consistent data, that all records are immutable, and that only invited members can read and write to the ledger. Setting up a new blockchain instance with a "Create a new Network" checkbox enabled identifies you as the founding member of the network. Other participants set up instances with this field unchecked and then join your network by sharing their PKI certificates with the founder. The founder's administrator can extend the network by adding the shared certificates to register the organizations as permitted members and enable them to participate in the designated channels. Other participants can also import member certificates when adding them in Create Channel dialogue.

This eBook uses an example of an ecosystem that includes a car manufacturer and a group of car dealers to illustrate how a blockchain network might be setup and blockchain application created for this ecosystem.

Oracle Blockchain Platform Console

The Oracle Blockchain Platform console helps you monitor the blockchain network and perform day to day administrative tasks.

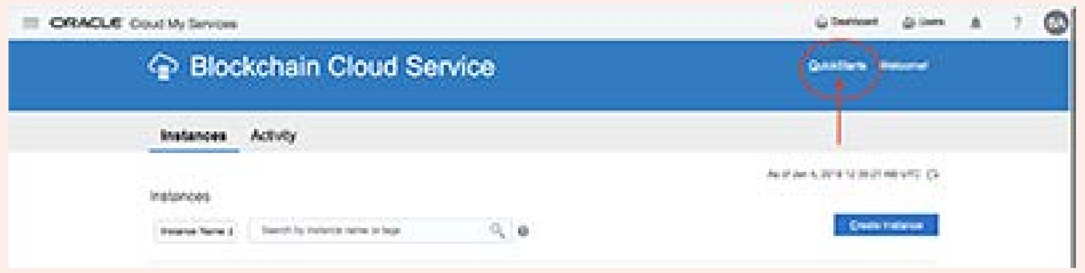
After you provision your OBP instance, all of the components and capabilities necessary to begin work on your blockchain network are available in the console Web UI or through the extensive REST API. You can use the console to perform tasks such as managing nodes, configuring network channels and policies, adding organizations and deploying chaincodes. You can also monitor and troubleshoot the network, view node status, view ledger blocks, and find and view log files. Detailed walkthrough of the Console tabs and capabilities is provided in Managing your Blockchain Network chapter.

Here we guide you through a few basic steps to set up the network and run included samples.

Each OBP instance in your network has its own console that their respective admins can use to manage their organization and monitor the blockchain networks that they are included in. User's role and their instances function in the network (founder or participant) determines the tasks you can do in your console. For example, if you are the Founder, you can edit the configuration of the orders used by the participants in the network.

Create a Blockchain Network (Founder)

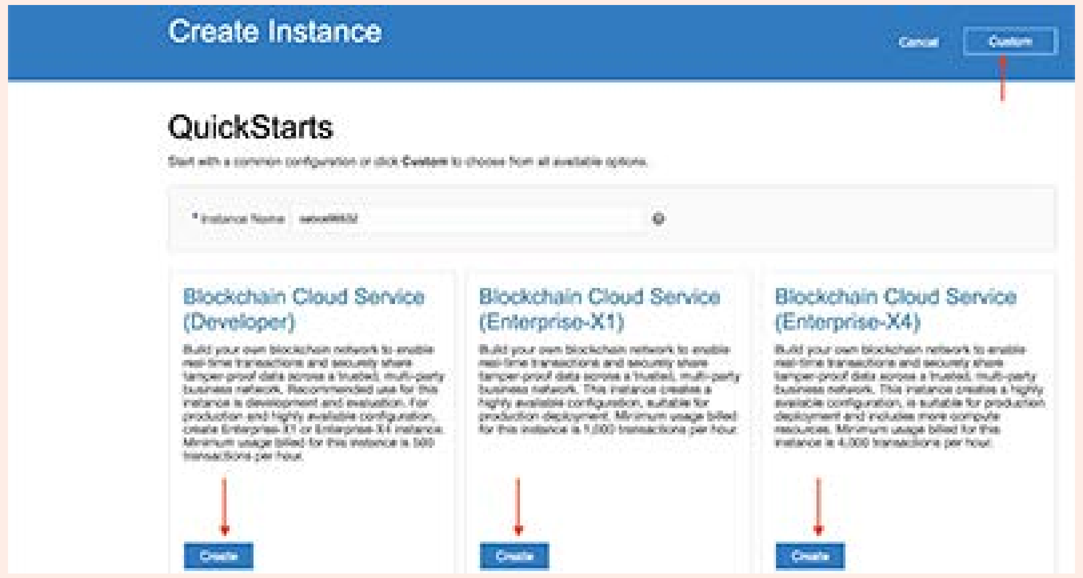
Developers can start creating the instance by selecting one of the network quick start templates. Developers can click on the "Quickstart" button to see the templates.



Select one of the three quick start templates by clicking on the **Create** button. The difference between Developer and Enterprise templates are detailed in the online documentation and summarized below:

- Enterprise templates include HA configurations for production where all components are replicated for resiliency
- Enterprise templates support zero-downtime managed patching/upgrades
- Developer template has a max of 7 peer nodes per instance while Enterprise templates have a max of 14 peer nodes per instance

The Enterprise X1 and X4 templates differ only in the number of cores allocated to VMs. While throughput depends on many factors, generally guidance is to use X1 template if you expect your transaction rates to be under 10K/hr (with payload sizes around a few hundred KB), otherwise choose X4 template. As the result of the HA and capacity differences, the minimum charges for each template are different.

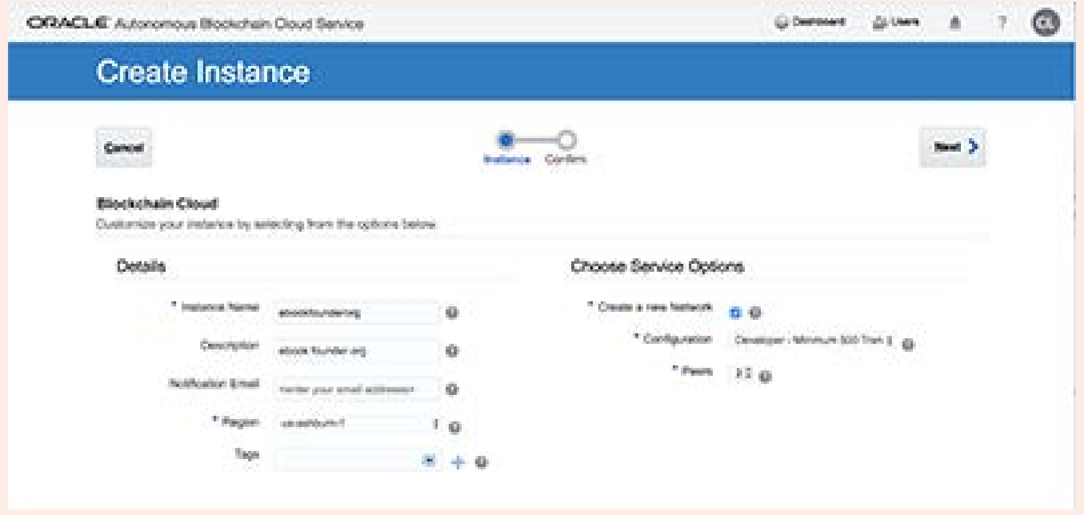

Create Instance
Cancel
Custom
QuickStarts
Start with a common configuration or click Custom to choose from all available options.
* Instance Name
Blockchain Cloud Service (Developer)
Blockchain Cloud Service (Enterprise-X1)
Blockchain Cloud Service (Enterprise-X4)
Create
Create
Create

Build your own blockchain network to enable real-time transactions and securely share tamper-proof data across a trusted, multi-party business network. Recommended use for this instance is development and evaluation. For production and highly available configuration, create Enterprise-X1 or Enterprise-X4 instance. Minimum usage billed for this instance is 500 transactions per hour.

Build your own blockchain network to enable real-time transactions and securely share tamper-proof data across a trusted, multi-party business network. This instance creates a highly available configuration, suitable for production deployment. Minimum usage billed for this instance is 1,000 transactions per hour.

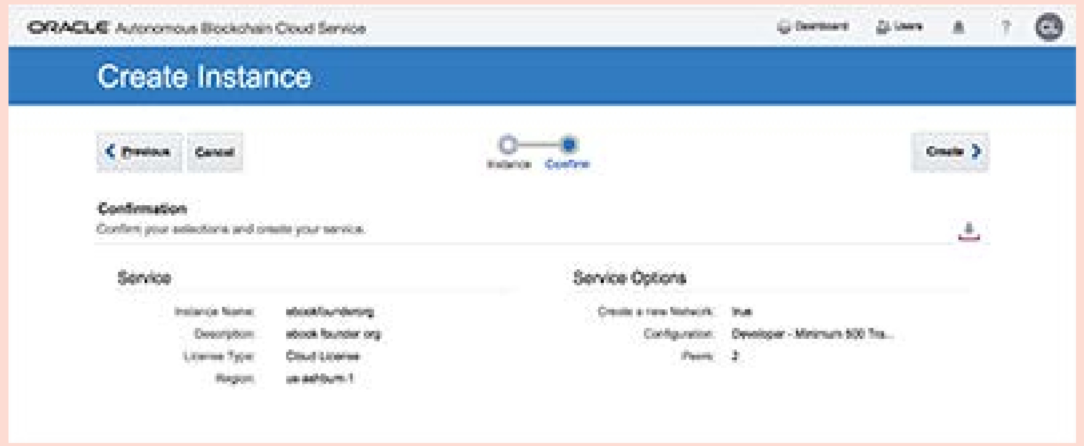
Build your own blockchain network to enable real-time transactions and securely share tamper-proof data across a trusted, multi-party business network. This instance creates a highly available configuration, is suitable for production deployment and includes more compute resources. Minimum usage billed for this instance is 4,000 transactions per hour.

Developers can also select **Custom** option to create their own instance configuration using the Create Instance dialogue shown below.


ORACLE Autonomous Blockchain Cloud Service
Dashboard
Users
Create Instance
Cancel
Instance
Confirm
Next
Blockchain Cloud
Customize your instance by selecting from the options below.
Details
Choose Service Options

* Instance Name	ebookfounderorg
Description	ebook founder org
Notification Email	<enter your email address>
* Region	us-ashburn-1
Tags	
* Create a new Network	✓
* Configuration	Developer - Minimum 500 Tran
* Peers	2

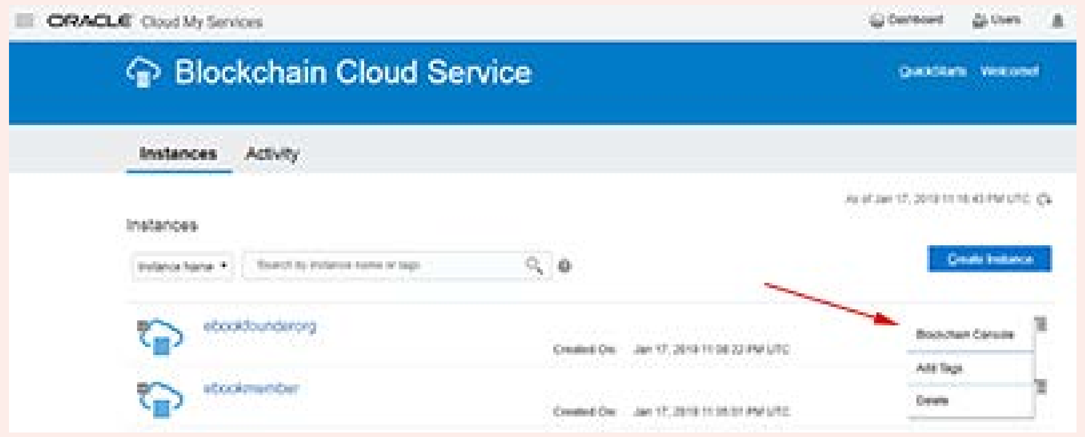
Click **Next** button in the top right corner to proceed to the confirmations screen and if all the details are correct, click **Create** button to submit the request.



Instance Name:	ebookfounderorg
Description:	ebook founder org
License Type:	Cloud License
Region:	us-ashburn-1
Create a new Network:	true
Configuration:	Developer - Minimum 500 Tra...
Peers:	2

The time to create an instance is typically under 10 minutes and you will receive a confirmation email once it's been created. You can also return back to the Blockchain Cloud Services dashboard and monitor the status there.

After the instance has been created, click on its menu icon on the right of the row and select "Blockchain Console" to bring up the Admin UI for your new blockchain instance.



As of Jan 17, 2019 11:18:43 PM UTC

ebookfounderorg	Created On: Jan 17, 2019 11:08:22 PM UTC
ebookmember	Created On: Jan 17, 2019 11:35:01 PM UTC

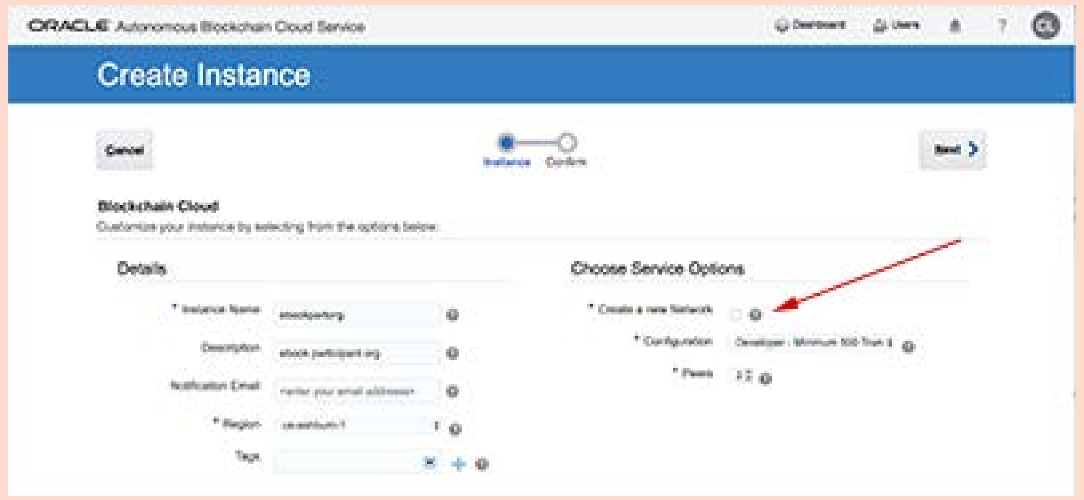
See chapter Managing your Blockchain Network below for a detailed walkthrough of the Console tabs and capabilities.

Adding Participating Organizations

A blockchain is more effective with multiple participating organizations. If you create additional instances, the founding member has the authority to add their organizations to the network. Provisioning is similar to the initial setup, with one difference – the participating organization doesn't need its own ordering service. If the instance you are creating is for an organization that will join an existing blockchain network, **do not check** the "**Create a new Network**" checkbox on the provisioning screen.

Free SDK →

Free Cloud Trial →


ORACLE Autonomous Blockchain Cloud Service

Create Instance

Blockchain Cloud
Customize your instance by selecting from the options below.

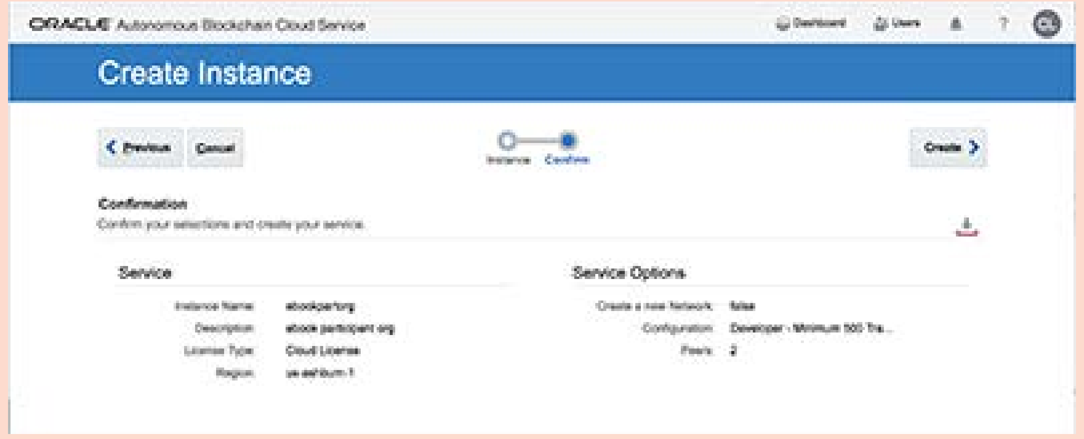
Details

* Instance Name	ebookpartorg
Description	ebook participant org
Notification Email	<enter your email address>
* Region	us-ashburn-1
Tags	

Choose Service Options

* Create a new Network	☐
* Configuration	Developer - Minimum 500 Tran...
* Peers	2

Similarly, click **Next**, verify your inputs, and then click **Create**.


ORACLE Autonomous Blockchain Cloud Service

Create Instance

Confirmation
Confirm your selections and create your service.

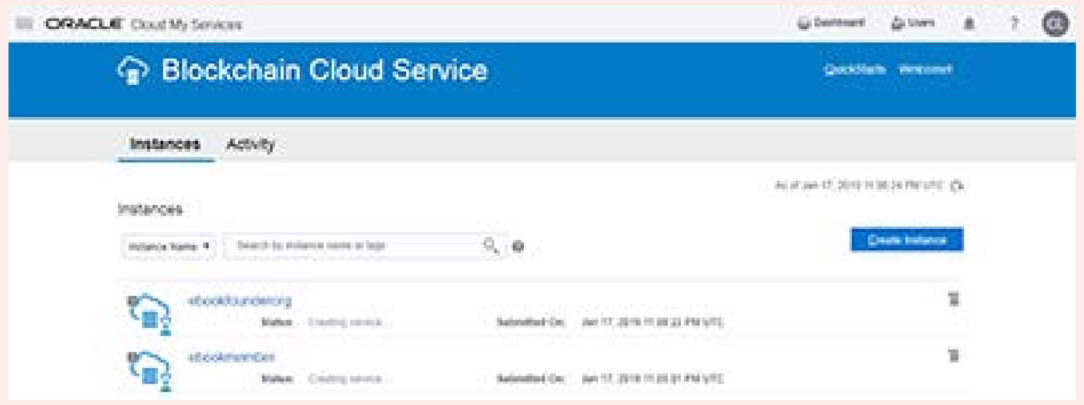
Service

Instance Name	ebookpartorg
Description	ebook participant org
License Type	Cloud License
Region	us-ashburn-1

Service Options

Create a new Network	false
Configuration	Developer - Minimum 500 Tra...
Peers	2

Upon return to the Blockchain Cloud Service dashboard you will see the instances with status "Creating service." You can also click on the Activity tab above to monitor these requests which should complete in about 10 minutes and generate an email notification.

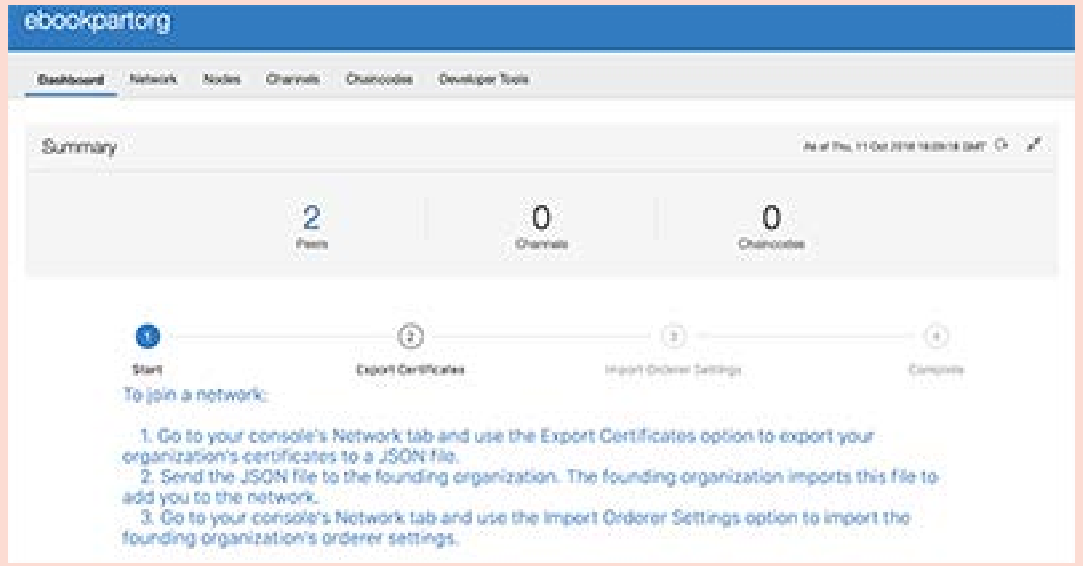

Blockchain Cloud Service

Instances

ebookfounderorg	Status: Creating service...	Submitted On: Jan 17, 2019 11:08:23 PM UTC
ebookmember	Status: Creating service...	Submitted On: Jan 17, 2019 11:09:01 PM UTC

Export Participant's PKI Certificates

Before the Founder or other members can add a new organization, the new member needs to export its PKI certificate as a JSON file. The console of each non-founding member loads a simple wizard to assist with joining the network on the Dashboard tab, which is the first view when you open the Blockchain Console. Once the instance has completed the tasks in the wizard, the Dashboard view changes to a normal set of health and traffic gauges. The wizard is shown in the following screenshot.


ebookpartorg

Summary

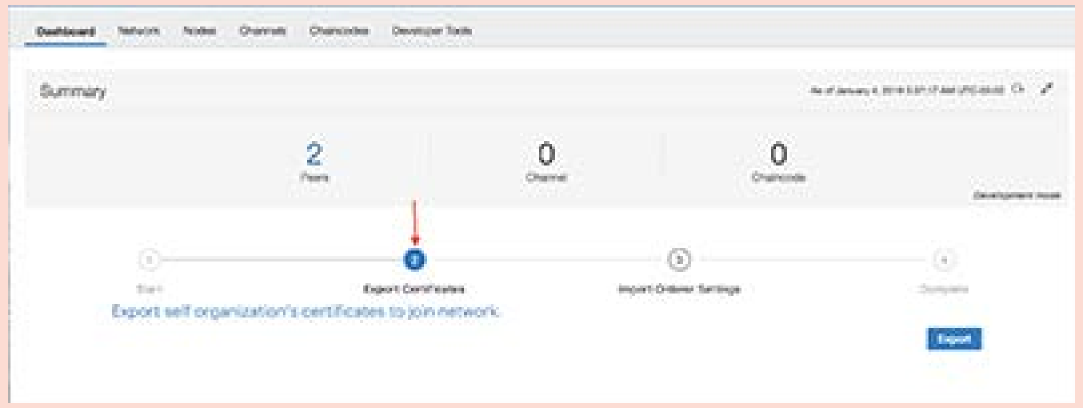
Peers	Channels	Chaincodes
2	0	0

To join a network:

1. Go to your console's Network tab and use the Export Certificates option to export your organization's certificates to a JSON file.
2. Send the JSON file to the founding organization. The founding organization imports this file to add you to the network.
3. Go to your console's Network tab and use the Import Orderer Settings option to import the founding organization's orderer settings.

Exporting of the certificates can be done automatically following the participant setup wizard or by exporting the certificate manually.

The developer can export the certificate automatically following the sequence in the participant setup wizard.


Summary

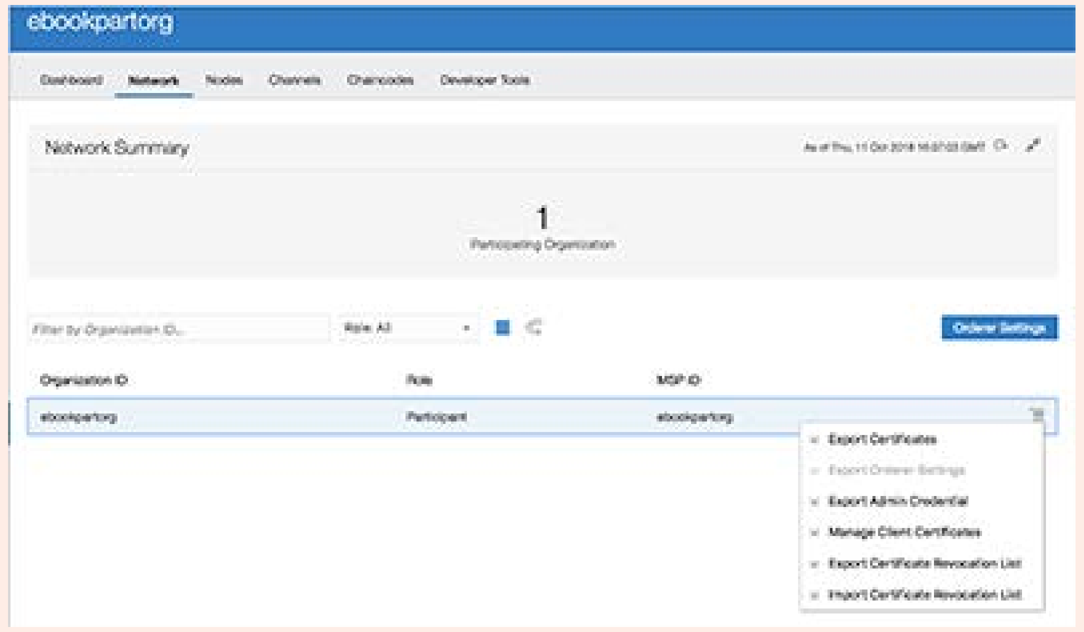
Peers	Channel	Chaincode
2	0	0

Export self organization's certificates to join network.

Free SDK →

Free Cloud Trial →

The developer can also manually export the certificate from the Network tab's view using the Actions menu pulldown on the far right of the organization's row as shown in thefollowing diagram.

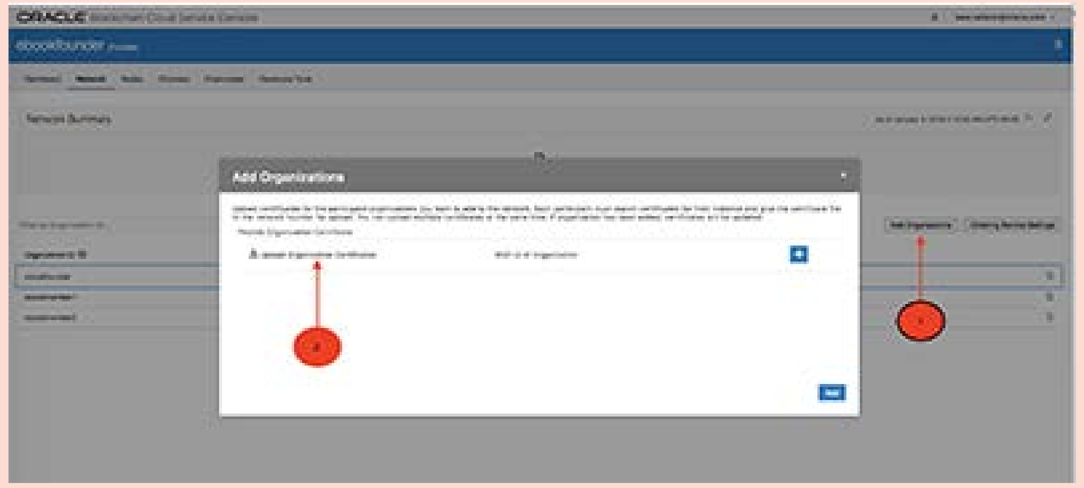


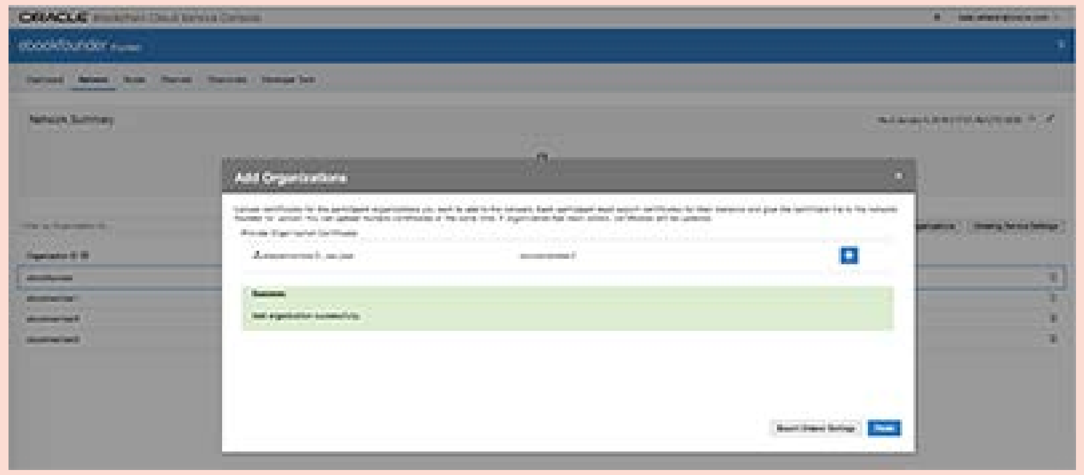
Organization ID	Role	MSP ID
ebookpartorg	Participant	ebookpartorg

Add Participant Organization to Founder

After the certificate of the participant organization has been exported, Founder can add participant organization by importing the certificate and validating the connection using Add Organization wizard on the Network tab of the Founder's console.

In the Founder console go to the Network tab and click **Add Organizations** button. In the wizard click **Upload Organization Certificates** control and select the JSON file exported in the Participant's wizard.

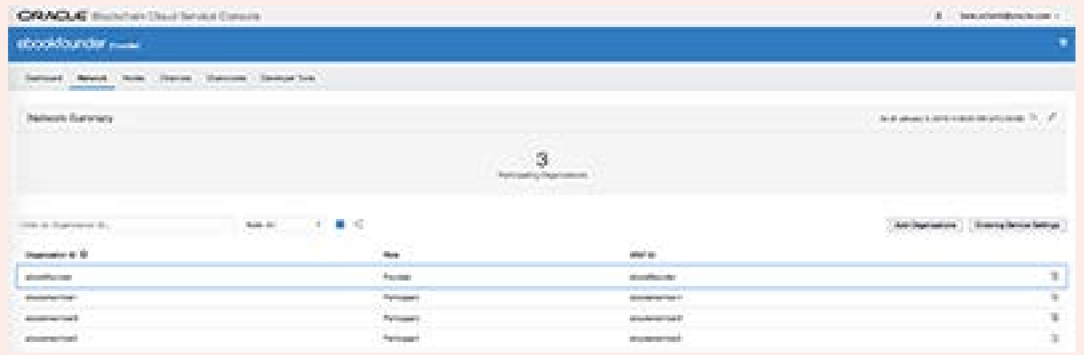




Multiple organizations can be added in one step using the **+** sign icon to add additional rows for importing multiple certificates.

Participants peer(s) can communicate with the network by connecting to the ordering service provided by the Founder. To obtain the orderer settings, after importing participant's certificate in to the founder network, click **Export Orderer Settings** button to save Founder's ordering service information in a JSON file. This will be imported later by the participating instance to complete the process of joining the network.

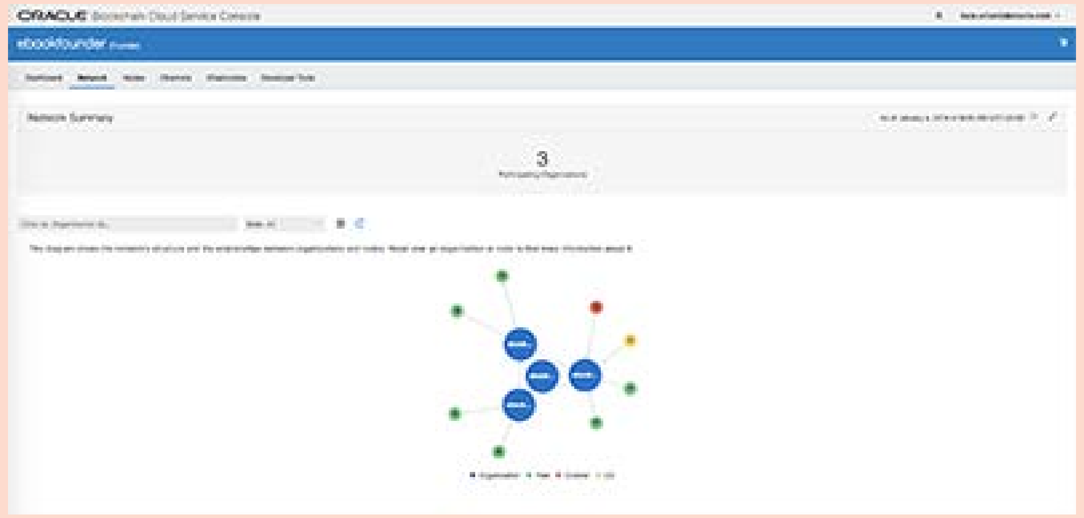
After you click **Finish** button, the participant organization is added to the Founder's network and you can see it listed in the Network table.



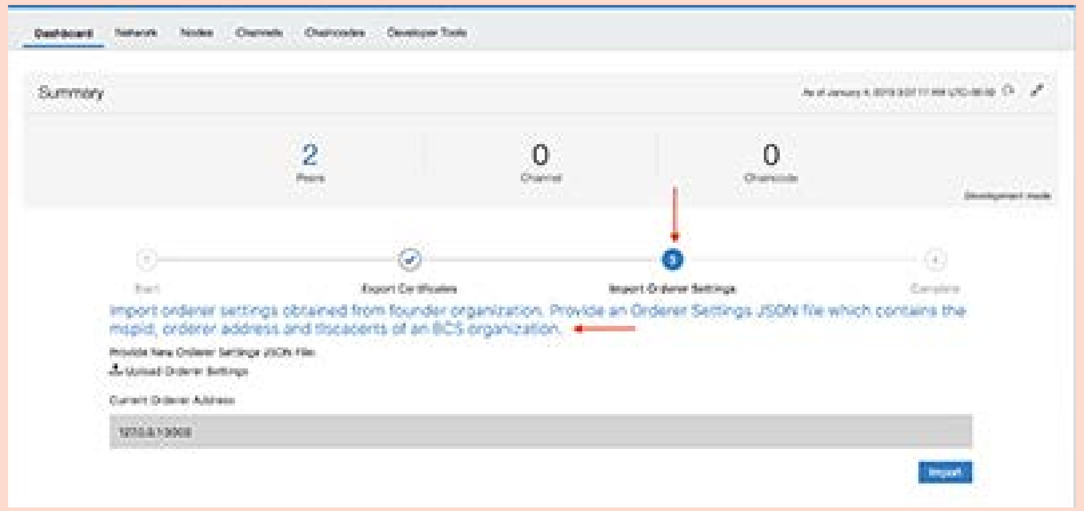
Free SDK →

Free Cloud Trial →

You can also switch from the table view to topology view by clicking on the topology icon ( ).

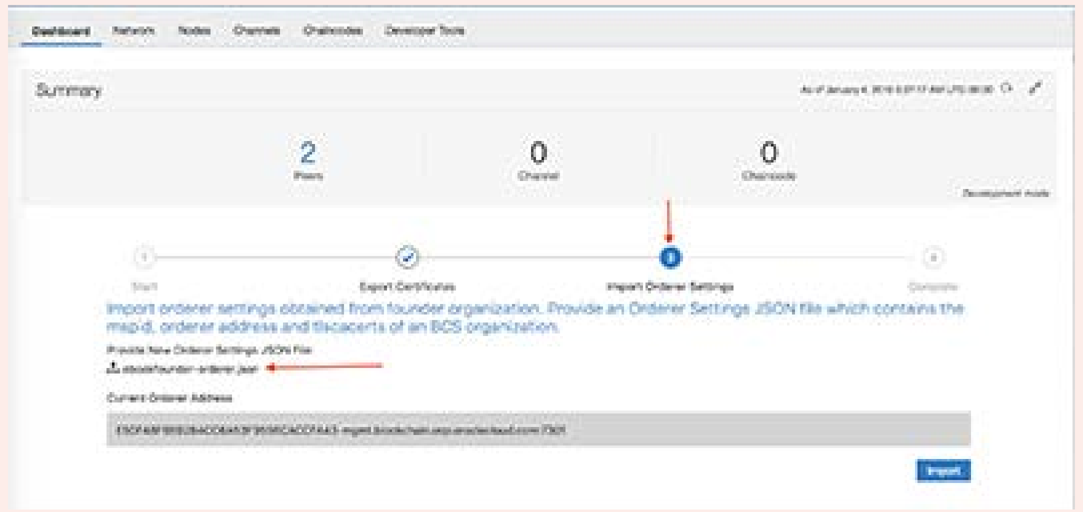


Now return to the participant's console and go to step 3 in the wizard to import the ordering service information you've exported from the Founder.



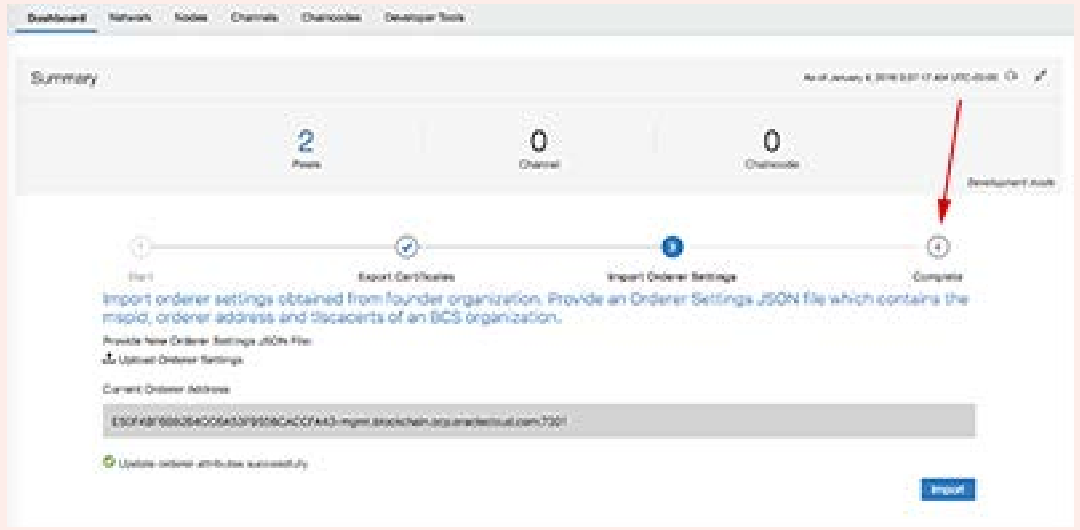
Import orderer settings obtained from founder organization. Provide an Orderer Settings JSON file which contains the mspid, orderer address and tlscacerts of an BCS organization.

Click on the Upload control and select the saved JSON file with the ordering service information, then click **Import** button.



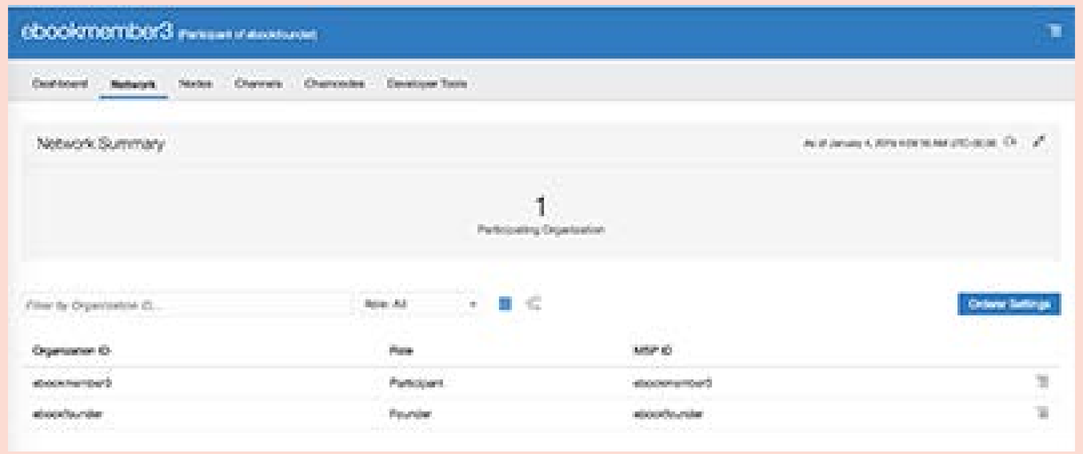
Import orderer settings obtained from founder organization. Provide an Orderer Settings JSON file which contains the mspid, orderer address and tlscacerts of an BCS organization.

After importing the ordering service information, click on step 4 in the wizard to complete the process and exit from the wizard to a regular dashboard view.

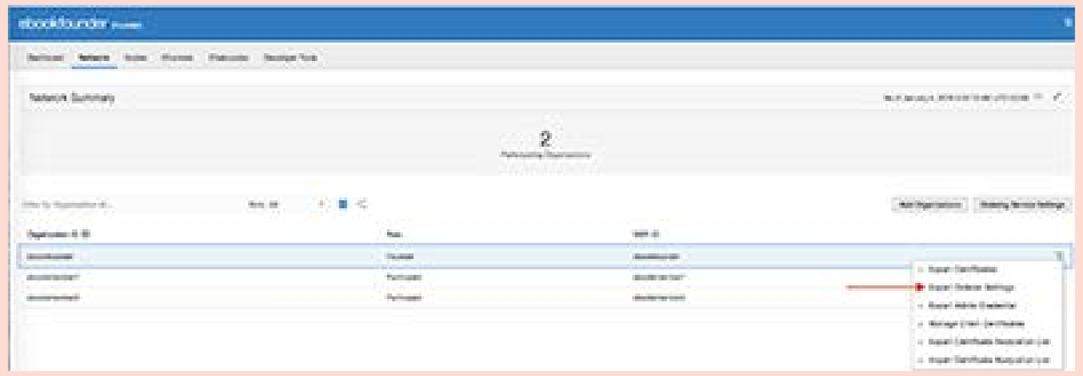


Import orderer settings obtained from founder organization. Provide an Orderer Settings JSON file which contains the mspid, orderer address and tlscacerts of an BCS organization.

In the Network tab of the participant's Console you can now see the entries for the Founder and your Participant instance.



By the way, if you have to export Founder's ordering service information again in the future, it is also available from the additional actions menu of the Founder in the Network tab.

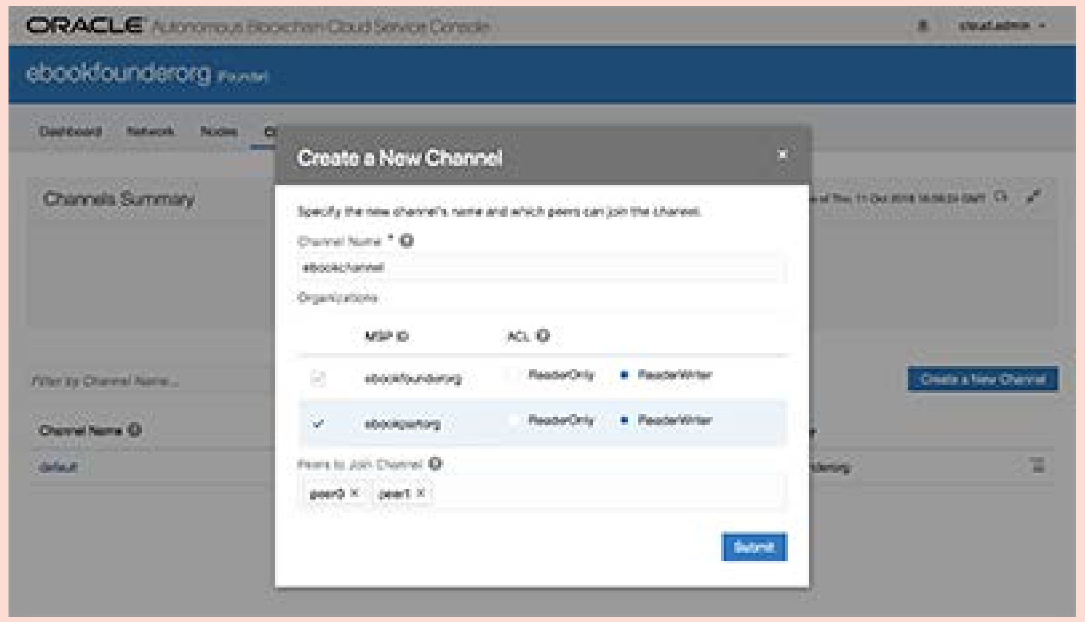


Creating Channels

The next step is connecting the peers on the channels. Channels is a sub network of the main blockchain network with isolated ledger and authorized members. Once authorized to join a channel, a member organization can add one or more of its peer nodes to the channel. Peers can participate in one or more channels, and for each channel have either read-only or read/write access to the ledger. Peers with read-only access can not run smart contracts for that channel, but do get the blocks and commit transactions in their copy of the channel's ledger. They can publish events and respond to block queries from Fabric client SDKs.

Note that a channel named "default" is created at provisioning time and all peers are automatically joined to that channel. You can continue to use the default channel or create new ones for certain transactions, chaincodes, and member organizations.

You could have a completely open network with all organizations and their peers belonging to one channel, or use a number of channels between groups of participants to enable the appropriate mix of shared and confidential actions. For example, you could keep a vehicle sales pricing information on a private channel confidential to the transaction participants, while keeping vehicle inventory status on a channel shared by all participating dealers.



Specify the new channel's name and which peers can join the channel.

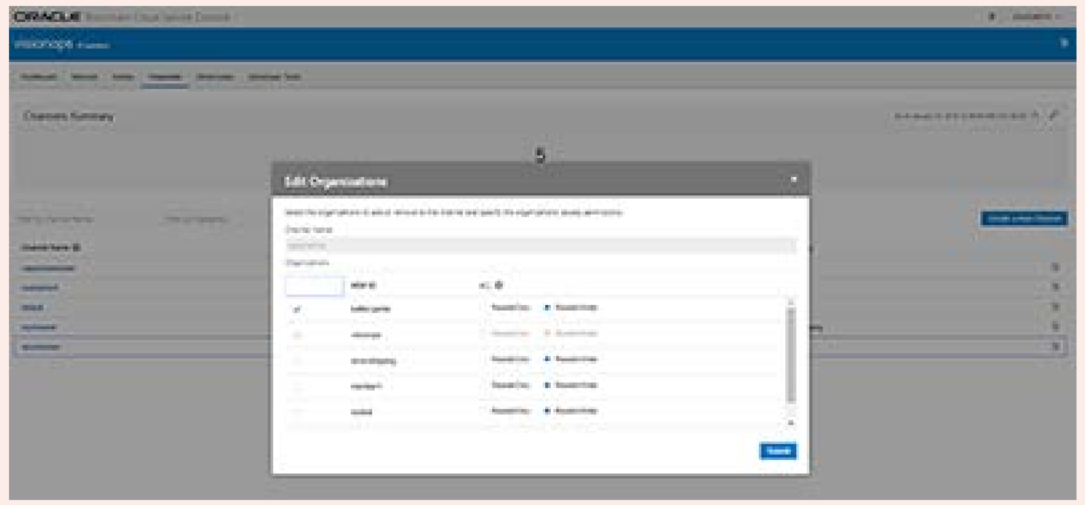
Channel Name *: ebookchannel

Organizations

	MSP ID	ACL
✓	ebookfounderorg	ReaderOnly ● ReaderWriter
✓	ebookpartorg	ReaderOnly ● ReaderWriter

Peers to Join Channel: peer0 × peer1 ×

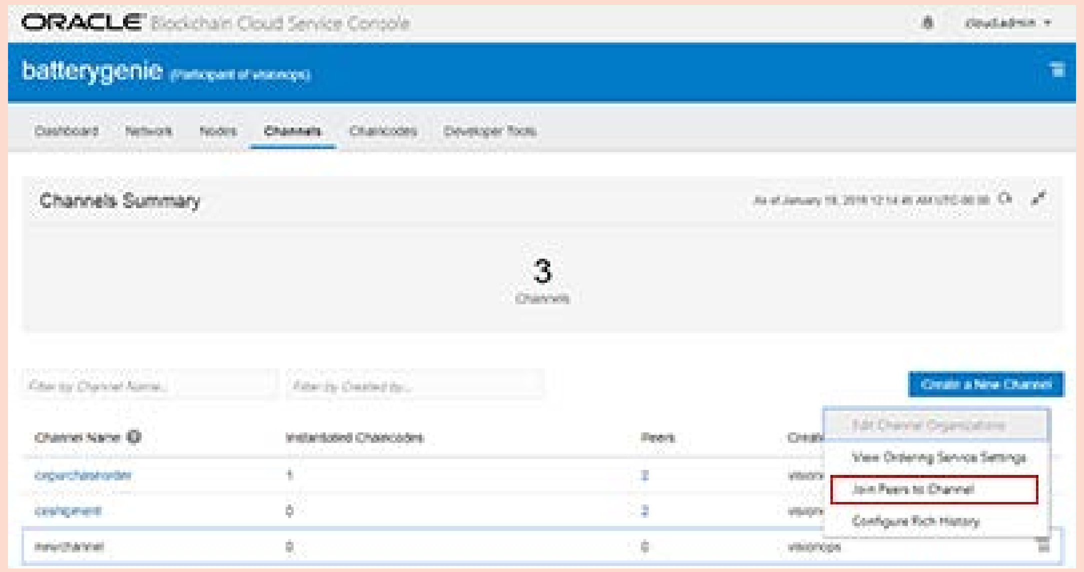
Like the other setup activities across the organizations, creating channels is a two-step operation. First you authorize a member organization to a channel and then each organization must explicitly join its peers. You can select the authorized members when creating a new channel using checkboxes in the **Create a New Channel** wizard (as shown above), or add a new members to an existing channel using **Edit Channel Organizations** option from the channel's additional actions menu on the Channels tab as shown below.



Free SDK →

Free Cloud Trial →

Once the organization has been added to a channel, the second step is to join its peers to that channel, and this must be done in that organization's console using **Join Peers to Channel** option from the channel's additional tasks menu on the Channel tab.

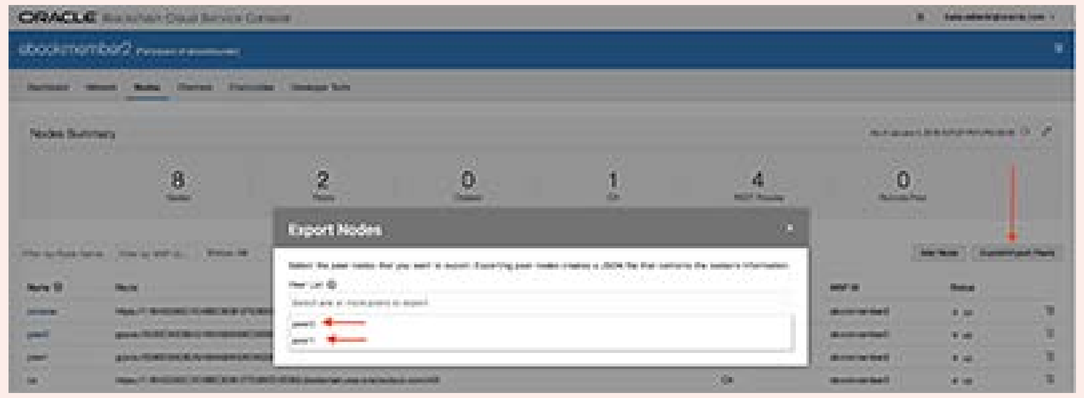


Channel Name	Instantiated Chaincodes	Peers	Creat
carpurchaseorder	1	2	vision
cashipment	0	2	vision
newchannel	0	0	visionops

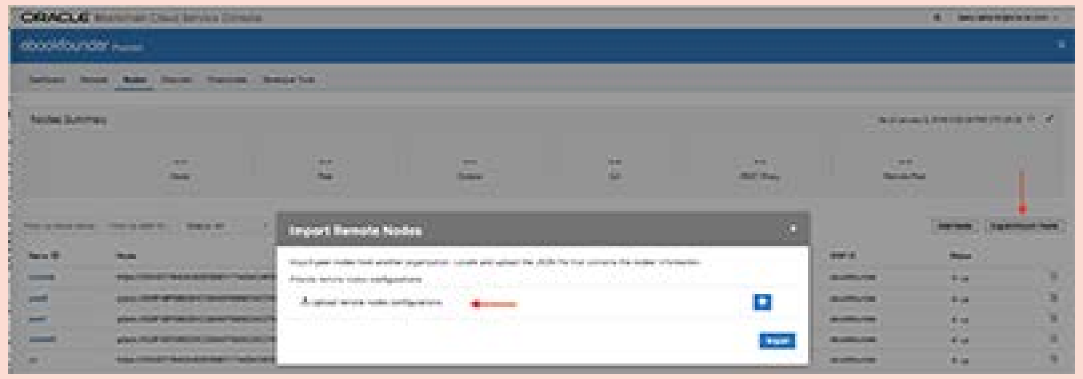
In our sample configuration, the founder's two peers are connected with participants (remote) peers on a channel.

An earlier practice of bilateral channels for each pair of participants is not considered very scalable and is best avoided. Note that since OBP 19.1.3 (based on Hyperledger Fabric 1.3) a new capability is available for **private collections** to maintain fine-grained confidentiality within a channel, where multiple collections of organizations can be defined within a channel and used for selectively sharing transaction data.

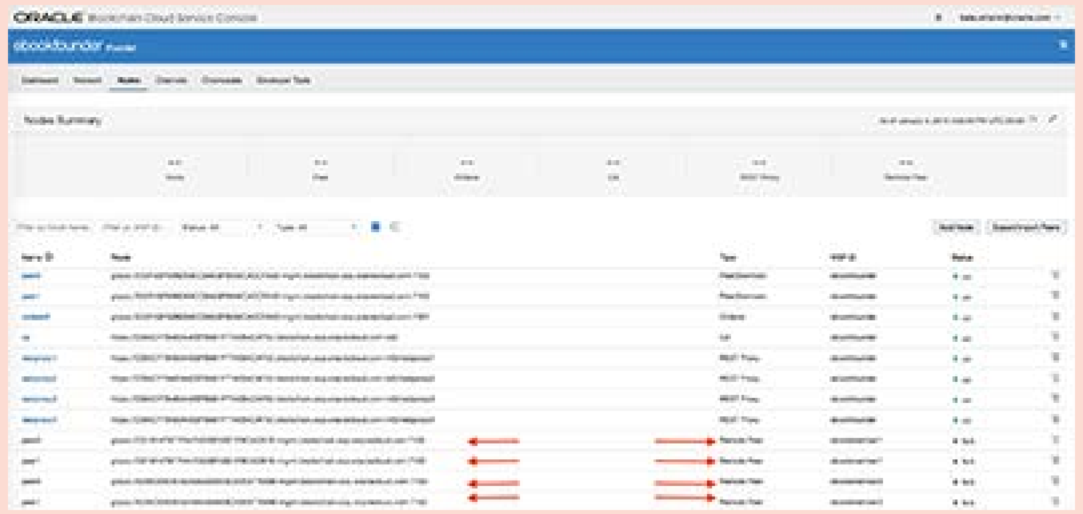
You can import and export information about other participant's peer nodes using **Export/Import** button on the Nodes menu. Export the remote peer nodes configuration from the participant's instance and then import in to the Founder's configuration:



Import the remote peer to the Founder's configuration:



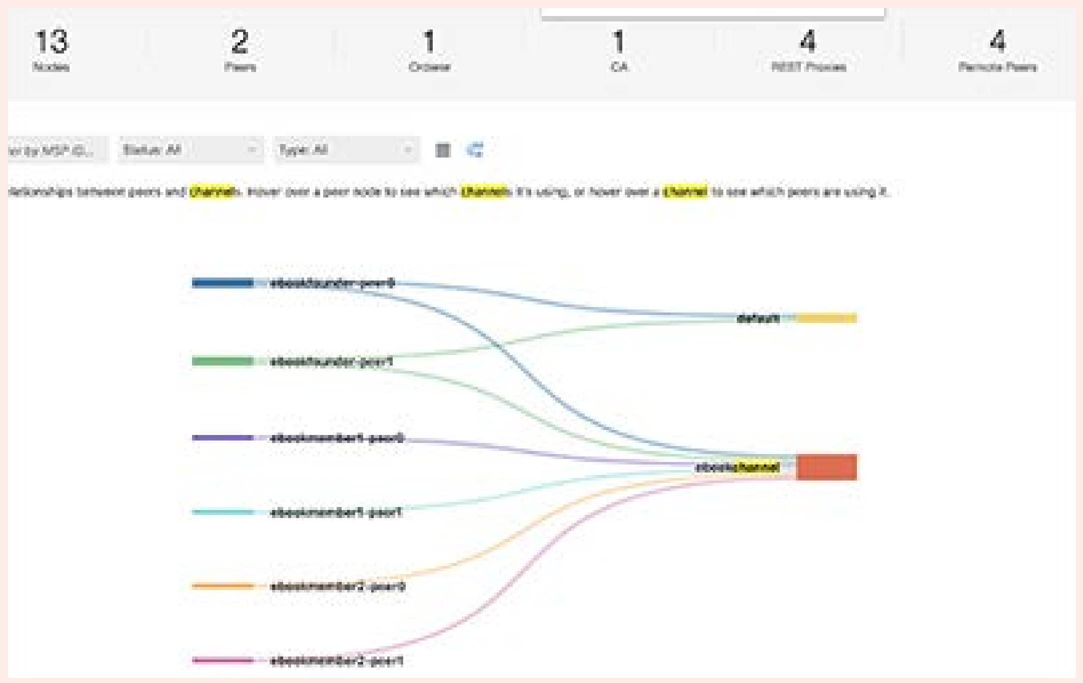
On the Nodes tab you can see remote peer nodes in the table below.



Free SDK →

Free Cloud Trial →

You can also see them associated with specific channels when you use the topology icon to switch to topology view. The fully provisioned blockchain network looks like this:



Relationships between peers and channels. Hover over a peer node to see which channels it's using, or hover over a channel to see which peers are using it.

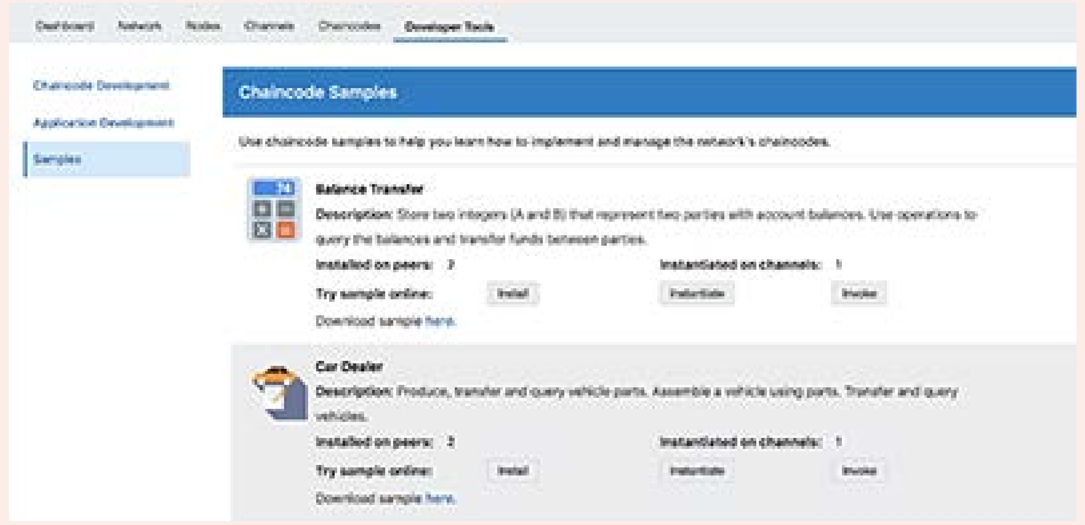
Developer Samples — Guided Walkthrough

Oracle Blockchain Platform (OBP) includes some chaincode samples to help you learn how to create, deploy, and invoke chaincode.

OBP console Samples page provides access to these samples to help you quickly get familiar with OBP and includes links to download complete chaincode and configuration artifacts for these samples. Two of the samples used in this eBook are:

1. Balance Transfer – A simple chaincode representing two parties with account balances and operations to query the balances and transfer funds between parties.

2. Car Dealer Sample – Chaincode to manage the production, transfer, and querying of vehicle parts; the vehicles assembled from these parts; and transfer of the vehicles. In this sample, a large automaker and its dealers and buyers have created a blockchain network to streamline its supply chain activities. Blockchain provides a distributed inventory ledger of vehicles and vehicle parts to enable visibility and transfer of vehicles and parts across this network. Developers can install, instantiate and invoke these samples from the Developer Tools tab under Samples.



Chaincode Samples

Use chaincode samples to help you learn how to implement and manage the network's chaincodes.

Balance Transfer
Description: Store two integers (A and B) that represent two parties with account balances. Use operations to query the balances and transfer funds between parties.
Installed on peers: 2 **Instantiated on channels:** 1
Try sample online:
Download sample here.

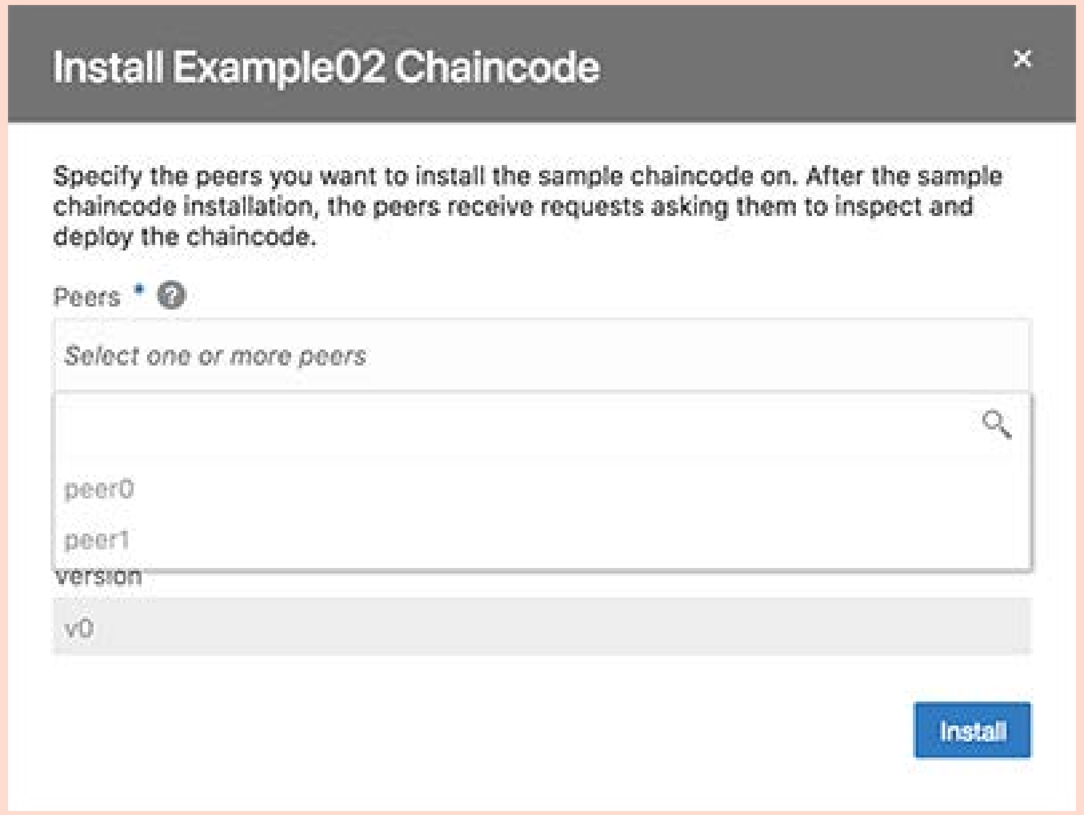
Car Dealer
Description: Produce, transfer and query vehicle parts. Assemble a vehicle using parts. Transfer and query vehicles.
Installed on peers: 2 **Instantiated on channels:** 1
Try sample online:
Download sample here.

Note the two areas below the samples: they provide an output display based on what chaincode returns (e.g., Success, or actual return values for queries), and a transaction log with detailed transaction flow steps and related messages.

To explore the samples, follow the Install, Instantiate, and Invoke steps for each sample. For more detailed explanation of the first two steps, see Developing and Deploying Chaincodes chapter below. Developers can also download the samples for learning chaincode programming.

Installing Samples

Click **Install** button and the chaincode will be copied to the peer nodes you specify. In the wizard you can select the peers to install it on.



Install Example02 Chaincode

Specify the peers you want to install the sample chaincode on. After the sample chaincode installation, the peers receive requests asking them to inspect and deploy the chaincode.

Peers *

Select one or more peers

peer0

peer1

Version

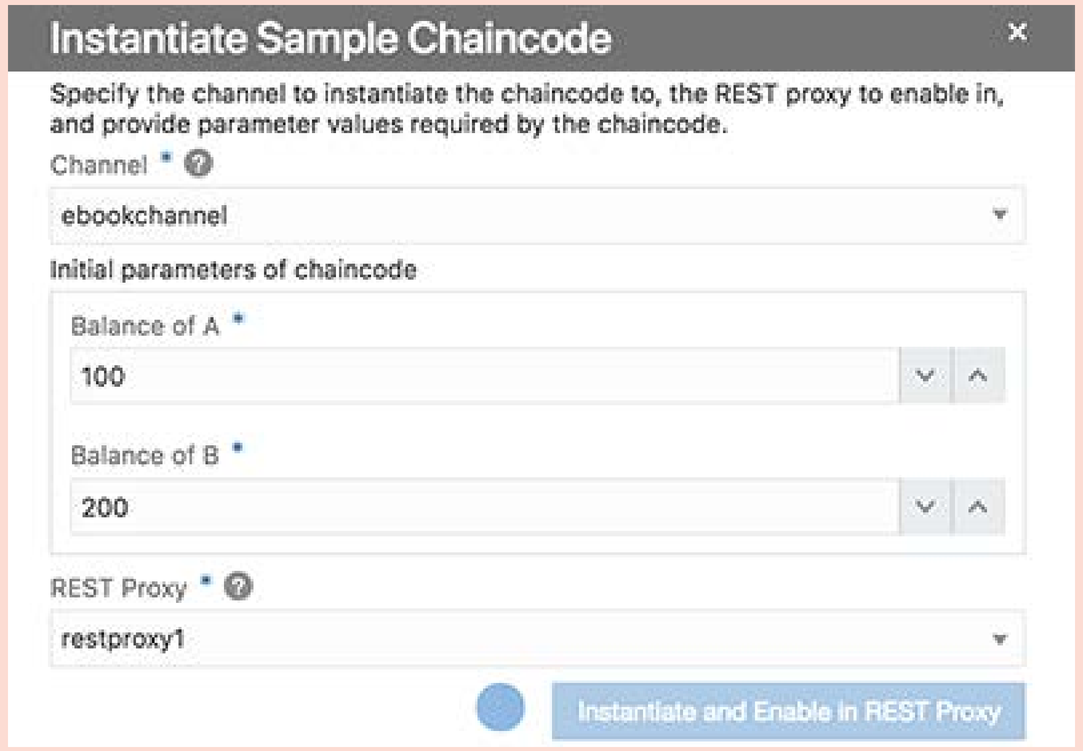
v0

Install

Instantiating Samples

Next step is to click Instantiate button. This multi–step process involves building (compiling) the chaincode, binding it to a channel, starting new chaincode execution containers initialized with the compiled chaincode and linked to the peer nodes of an organization, and finally invoking the chaincode's Init() function to initialize any ledger values (this could be a null function if developer so chooses.) This step also automates the process of creating a REST end point for this chaincode in the REST proxy. In the wizard you can select the channel to use, input parameters for the ledger initialization (optional depending on the chaincode), and rest proxy to use for the REST end point.

When instantiating BalanceTransfer sample, the chaincode expects to initialize the ledger with initial account balances, so you need to specify the input values for A and B accounts. However, in the Car Dealer sample, the ledger is not initialized and so no input fields are provided. Instead, the first action selected in the Invoke Chaincode wizard must be “Produce vehicle part”, which populates the ledger with initial data. Note that the instantiation process involves compiling the chaincode, creating a new execution container, loading the binary, and executing the chaincodes Init() function – this can take a couple minutes, so you may not see the Instantiated count updated immediately. It is safe to exit the wizard after this step and refresh the Samples page later to see if the chaincode instantiation count has been updated. You can also check it under Chaincodes tab or under Channels tab.



Instantiate Sample Chaincode

Specify the channel to instantiate the chaincode to, the REST proxy to enable in, and provide parameter values required by the chaincode.

Channel *

ebookchannel

Initial parameters of chaincode

Balance of A *

100

Balance of B *

200

REST Proxy *

restproxy1

Instantiate and Enable in REST Proxy

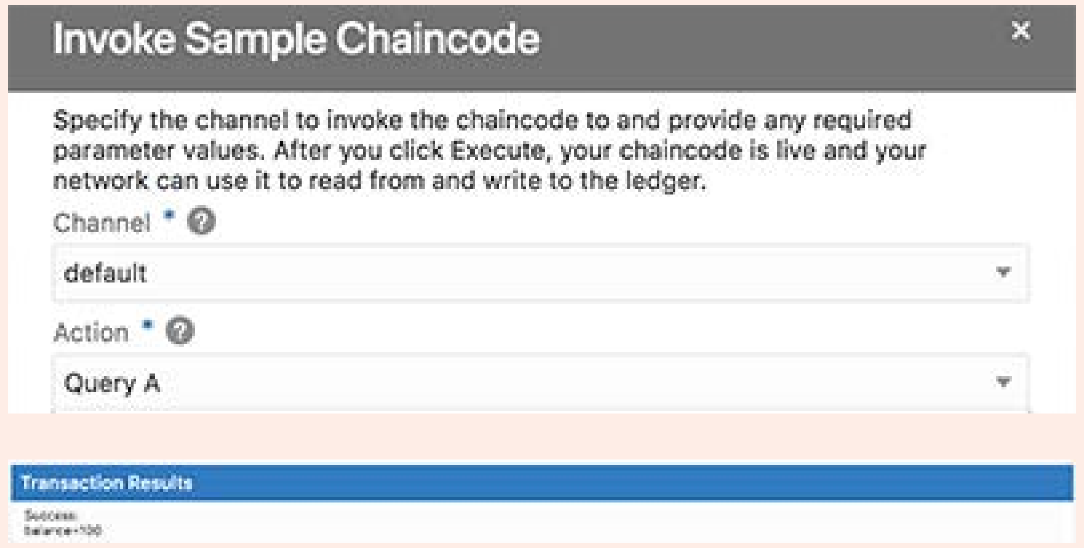
Free SDK →

Free Cloud Trial →

Invoking Samples

After chaincode has been instantiated, we are ready to execute the chaincode on the network peers of an organization. When you get to the Invoke step, the wizard offers a selection of actions to choose for that specific chaincode and input fields for any parameters they require.

In the BalanceTransfer sample, the ledger maintains values for two accounts denoted A and B, and the chaincode provides actions to transfer funds between them and query the balance of each account.


Invoke Sample Chaincode

Specify the channel to invoke the chaincode to and provide any required parameter values. After you click Execute, your chaincode is live and your network can use it to read from and write to the ledger.

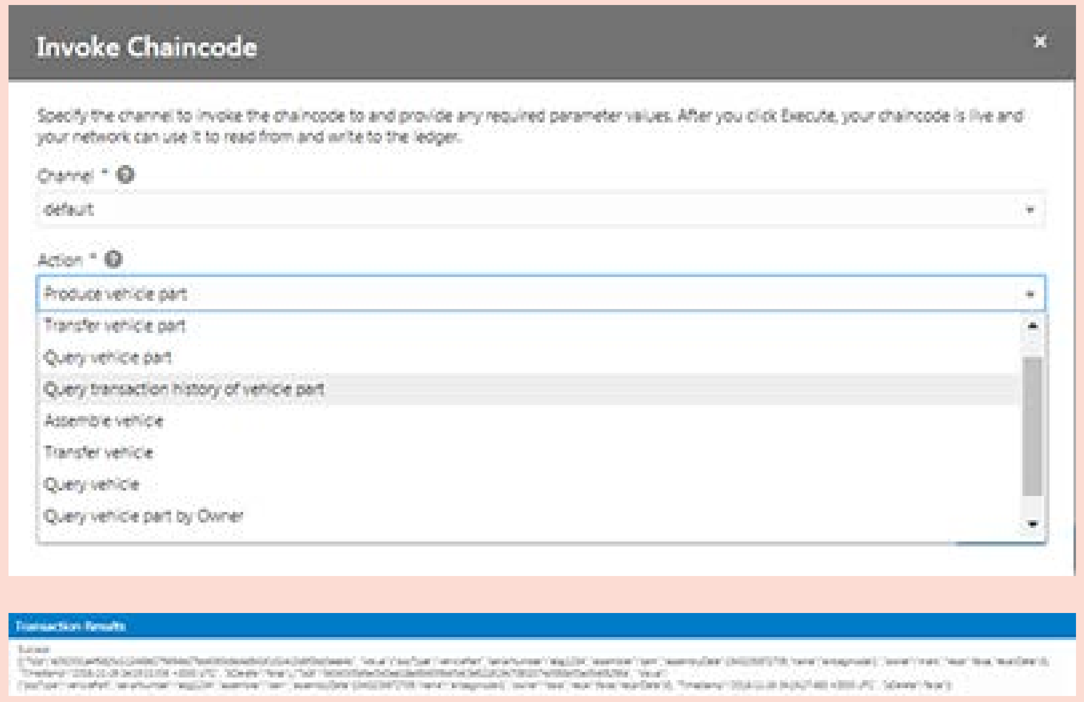
Channel * : default

Action * : Query A

Transaction Results

```
Success
balance=100
```

In the Car Dealer sample, the ledger maintains the inventory of vehicle parts and vehicle assemblies. The chaincode provides actions to add parts or vehicles to the inventory, transfer these to other owners, assemble parts into vehicles, and query the ownership or transaction history of parts and vehicles.


Invoke Chaincode

Specify the channel to invoke the chaincode to and provide any required parameter values. After you click Execute, your chaincode is live and your network can use it to read from and write to the ledger.

Channel * : default

Action * : Produce vehicle part

Transaction Results

```
Success
[illegible]
```

After running these transactions, let's review the rest of the Console to understand the information it provides about the blockchain network and the latest transactions.

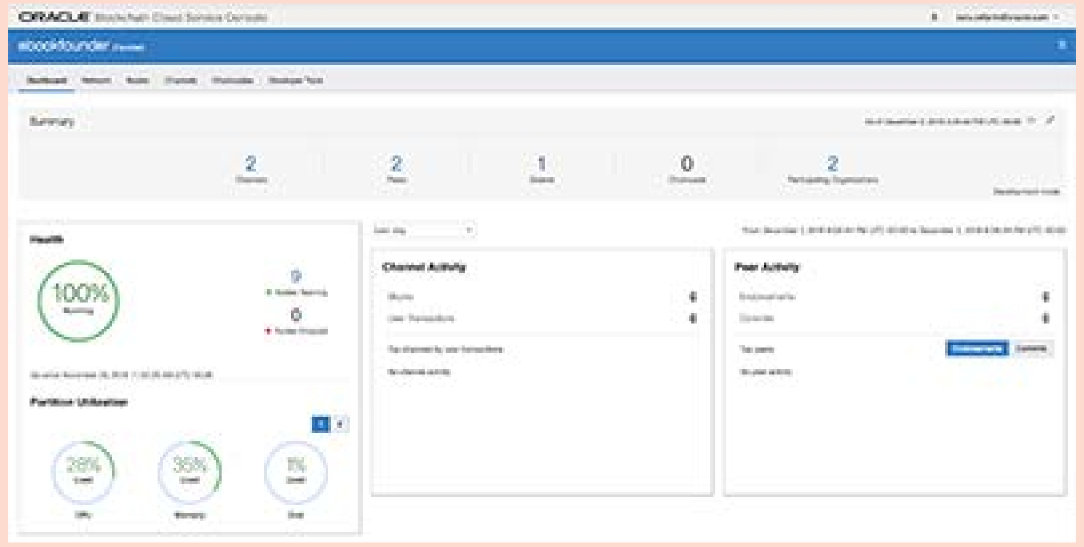
Managing your Blockchain Network

Oracle's blockchain console provides all of the tools necessary for you and the other member organizations to monitor and manage blockchain network participation. Each organization will have their own console UI where they can monitor network health, add organizations (if authorized), view node status, examine channels and ledger activity, and manage chaincodes.

Dashboard Tab

The Dashboard tab provides a summary of the network health and transaction metrics, with a banner summarizing the number of operating components, such as peers, channels, and chaincodes. A health monitor displays the number and percentage of running nodes, while the transaction charts show the most used channels and peers during the selected time frame. Note that Peers charts can be switched to show endorsement or commit transaction counts.

After the sample transactions have been executed, the transaction metrics on the default channel and on some of the peers should've increased to reflect the most recent transactions.



Information Provided in the Dashboard:

1. Health of the Network:

- Percentage and number of running and stopped nodes (any stopped nodes can be started from Nodes tab.)
- Partition Utilization showing CPU, Memory, and Disk used in each of the two OBP manager VMs. You can toggle the view between the partitions by clicking **1** and **2** buttons.

2. Channel Activity:

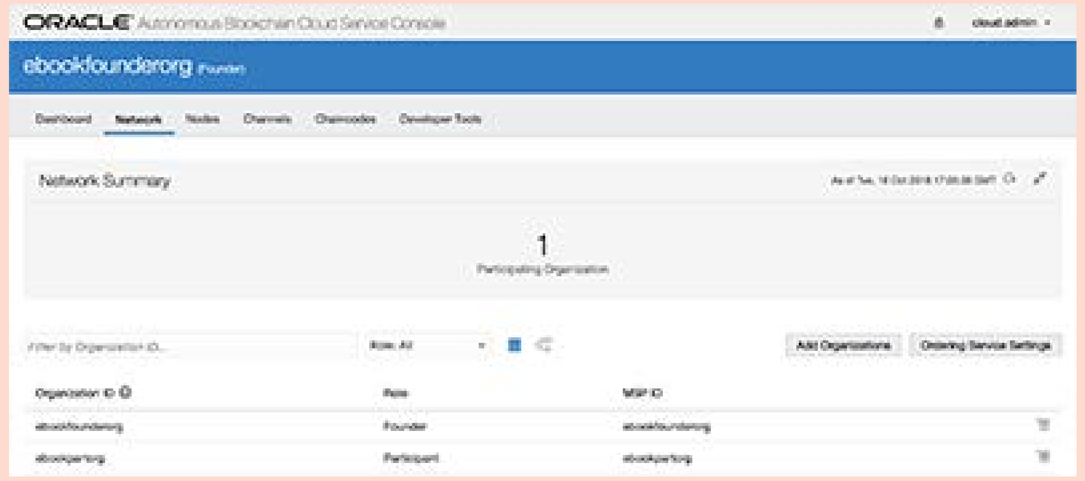
- Number of blocks that have been created.
- Number of transactions that have been executed and number of blocks created. Note that system transactions, e.g., configuration changes, chaincode deployment or update will increase the number of blocks, but will not show up in user transactions. In addition, a block can have multiple transactions based on ordering service configuration settings.
- Graphic depiction of the most active channels.

3. Peer Activity:

- Number of endorsement and commits completed by the network's peer nodes.
- Graphic depiction of the most active peer nodes with success and failure metrics, and a toggle between Endorsements and Commits.

Network Tab

The network tab lists all of the organizations currently participating in your blockchain network. Network views are available in tabular (☰) or graphical topology (⧉) formats and provide additional details on each organization's role. You can also use **Add Organizations** button on this tab to launch a wizard used to add and remove organizations from your network.



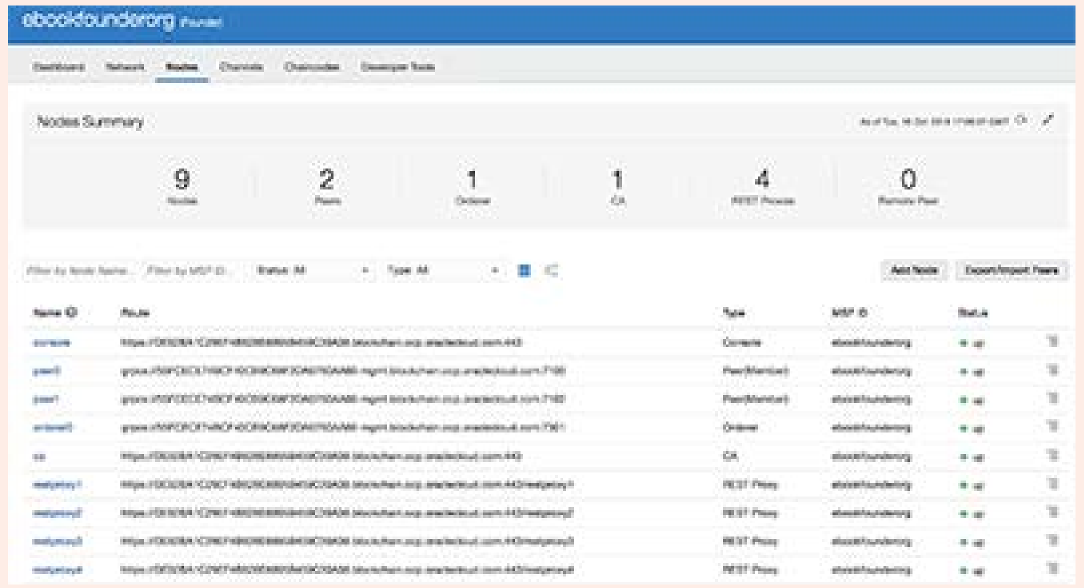
Organization ID	Role	MSP ID
ebookfounderorg	Founder	ebookfounderorg
ebookpartorg	Participant	ebookpartorg

Free SDK →

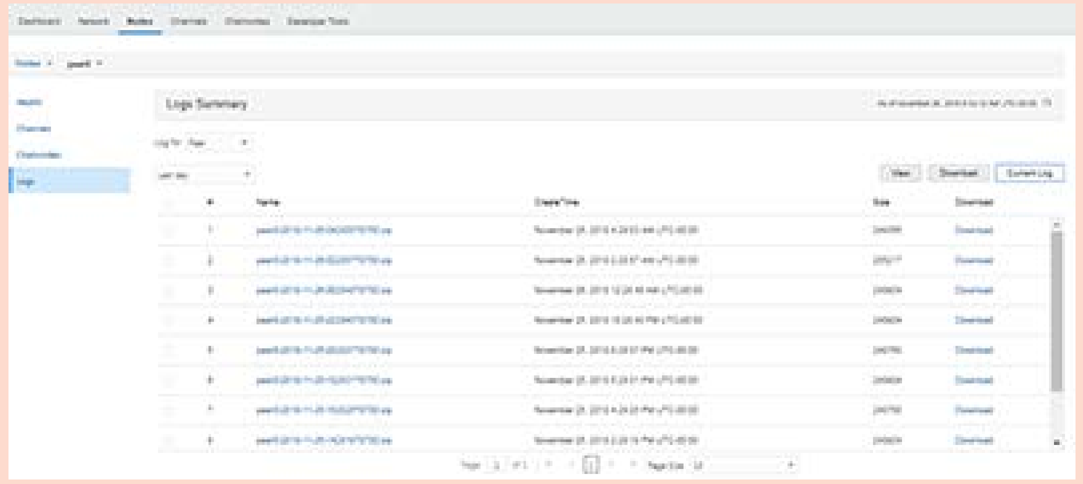
Free Cloud Trial →

Nodes Tab

The Nodes tab has a summary banner displaying counts of each node type, including your peers, orderers, certificate authorities, REST proxies, and remote peers. This tab also lists details for each node, including the full name, nod type, and current status. You can use the Actions menu (☰) on the right-hand side of each node's row to start, stop, and restart nodes, join a node to a channel, or edit its configuration. **Add Node** button enables you to create additional Peer nodes, and **Export/Import Peers** button provides access to wizards used to export and import peer node information between instances.



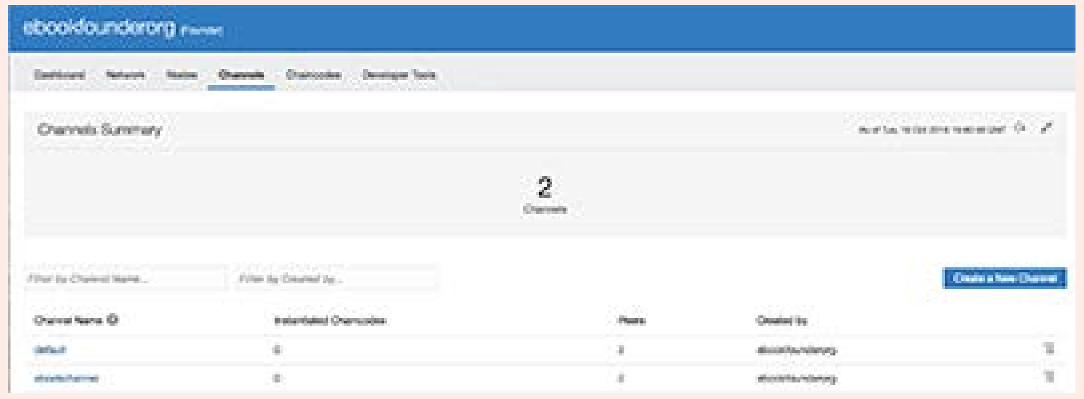
Clicking on the node name itself brings up a node-specific set of views, including health gauges and Logs view. You can check peer node logs by going to Nodes tab, double clicking on a peer node (e.g., peer0) and selecting Logs from the nav bar on the left. In the Logs select Peer or Chaincode logs to view and then click **Current Log** button to see the entries related to the sample transactions you've just executed. The logs are rotated (snapped and zipped) at midnight UTC daily, and by setting the time window you can view and download the logs for previous days.



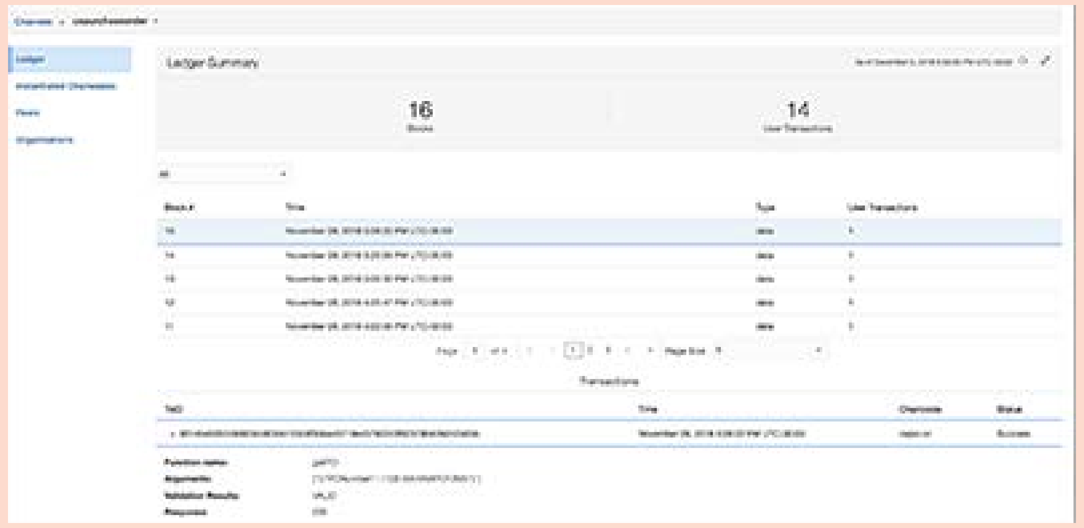
Channels Tab

The Channels tab provides a view of each channel, which is a blockchain subnet, including its creator organization, the number of peers and instantiated chaincodes. Note that only the channels you created or have been authorized to join will be visible in your instance. If you don't see a channel you expect to see listed, check in its creator's console (Edit Channel Organization on the Actions menu in the Channels tab) if your organization has been authorized as indicated by the checkbox.

You use this tab to add new channels and upgrade chaincodes on a channel. This tab also enables you to drill down on the channel to view detailed ledger activity for each channel.



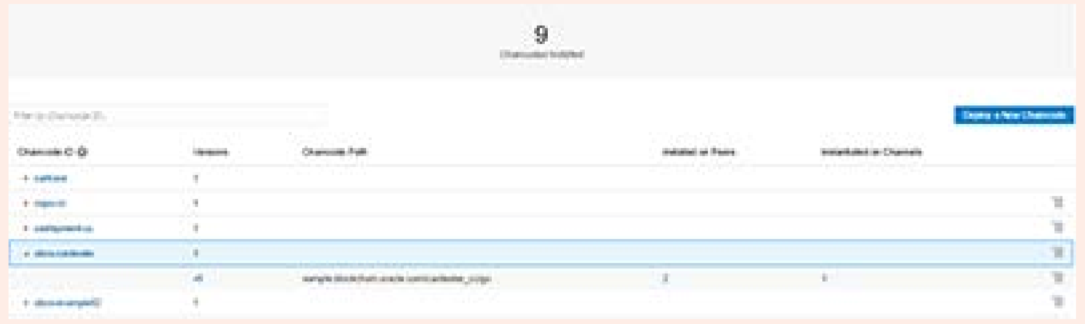
Drill down on the **default** channel to use ledger browser view to explore the added blocks. Highlighting a block will show its transactions below, and expanding the triangle icon to the left of a transaction will show its details.



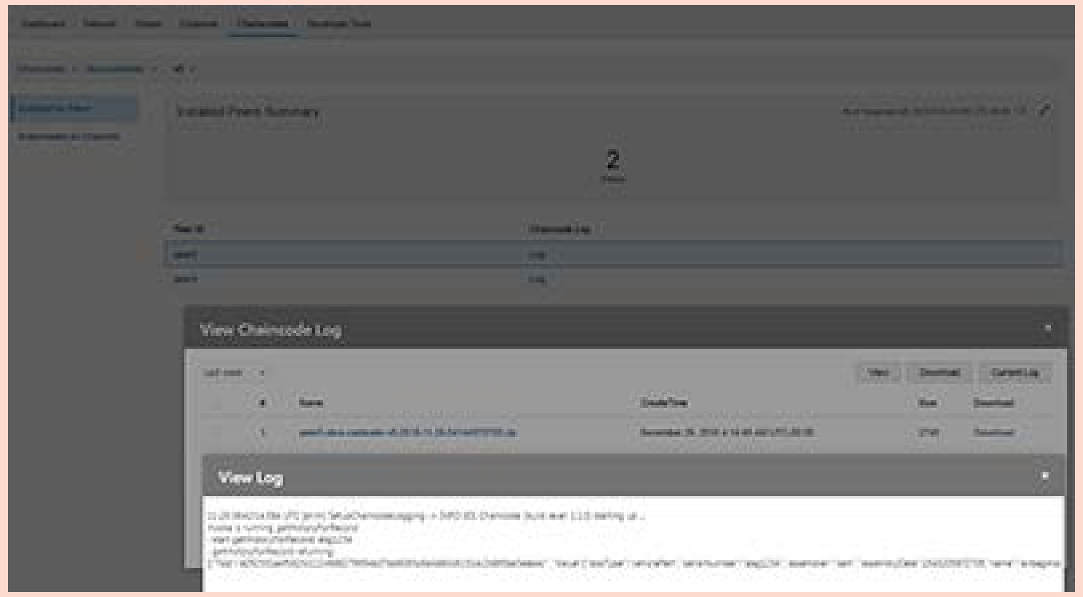
The drill down page also provides access to view the other information related to this channel: instantiated chaincodes, peer nodes and organizations that have been authorized on this channel.

Chaincodes Tab

The Chaincodes tab lists the chaincodes installed on your network and their versions. You can use this tab to determine which peers and channels have chaincodes installed and to deploy new chaincodes. The Install chaincode button provides access to two wizards: Quick Deploy and Advanced. The former is a single dialogue wizard that handles chaincode installation, instantiation (building and binding to a channel), and creation of a REST end point through the REST proxy—all with very few inputs from the user. The Advanced wizard takes you through a three-step dialogue for these operations, providing an opportunity to specify detailed settings for the peers to install on and define chaincode's endorsement policy, transient map, and private collections as well as to choose the REST proxy to use to expose this chaincode.



For detailed chaincode execution logs, drill down on **v0** for **obcs-cardealer** chaincode, then click on Logs link for any of the listed peers, and on Logs view click the **Current Log** button or one of the rotated archived logs and scroll down to see the entries related to most recent transactions.



Free SDK →

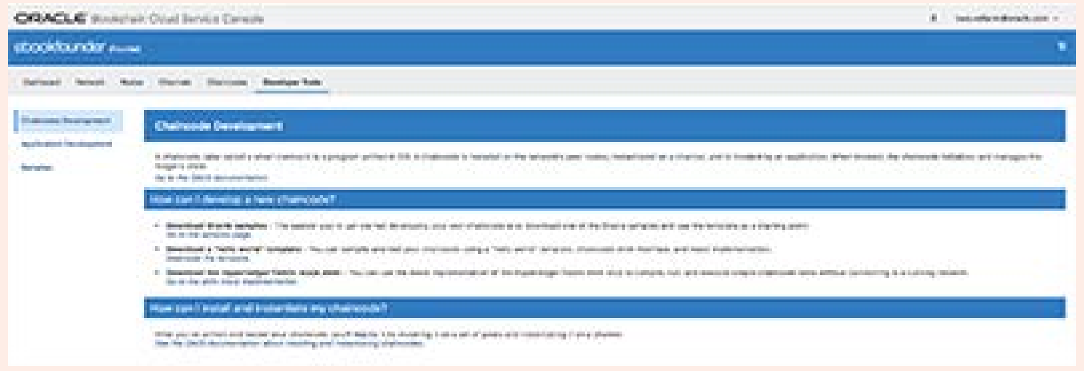
Free Cloud Trial →

Developer Tools Tab

The developer tools are used to learn how to develop chaincode, develop applications on Oracle Blockchain Cloud Service. This tab also provides samples to install, instantiate & test the chaincode samples Oracle Blockchain Cloud Service Console.

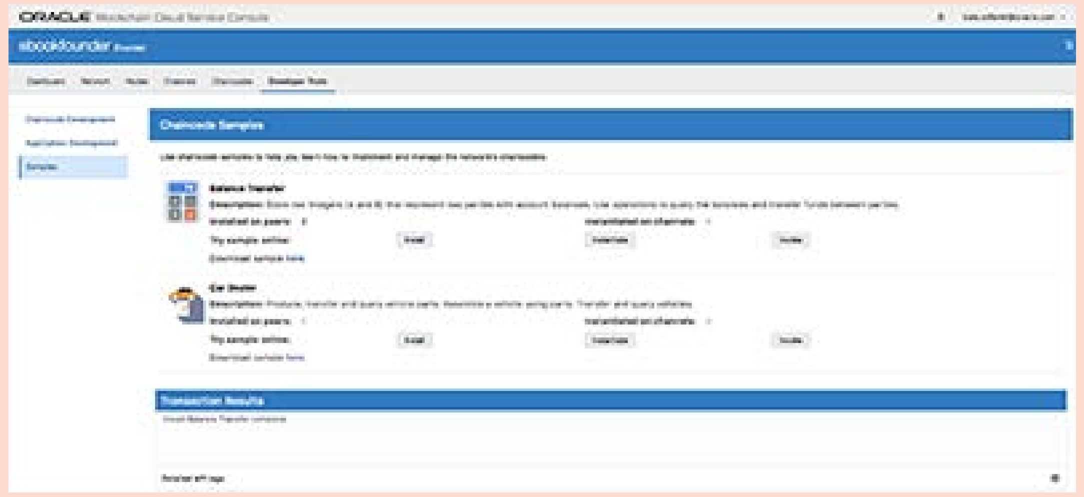
Chaincode Development

This includes chaincode samples written in Go and Node.js to help you learn how to implement and manage your network's chaincodes.


ORACLE Blockchain Cloud Service Console
Chaincode Development
How can I develop a new chaincode?
How can I install and instantiate my chaincode?

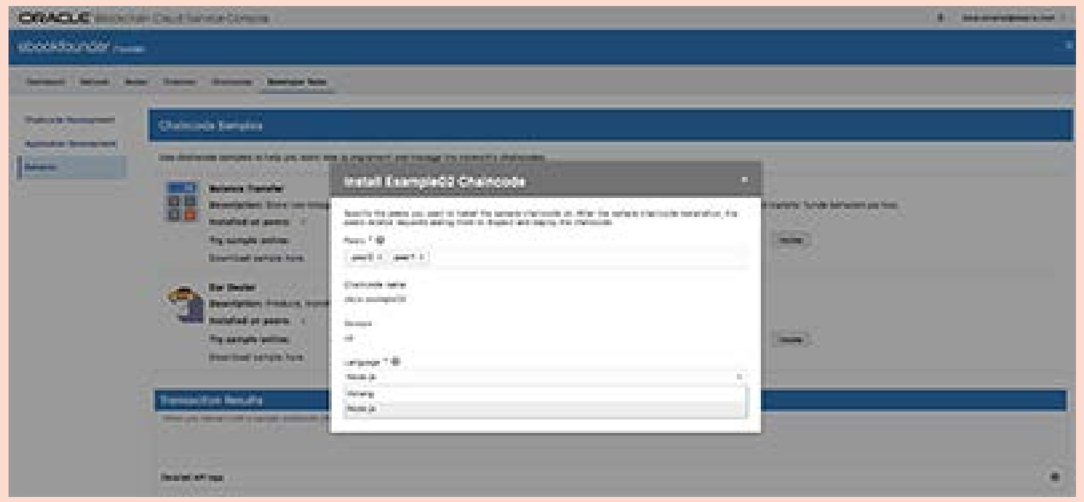
Samples

This page has options to install, instantiate & test the sample chaincode.



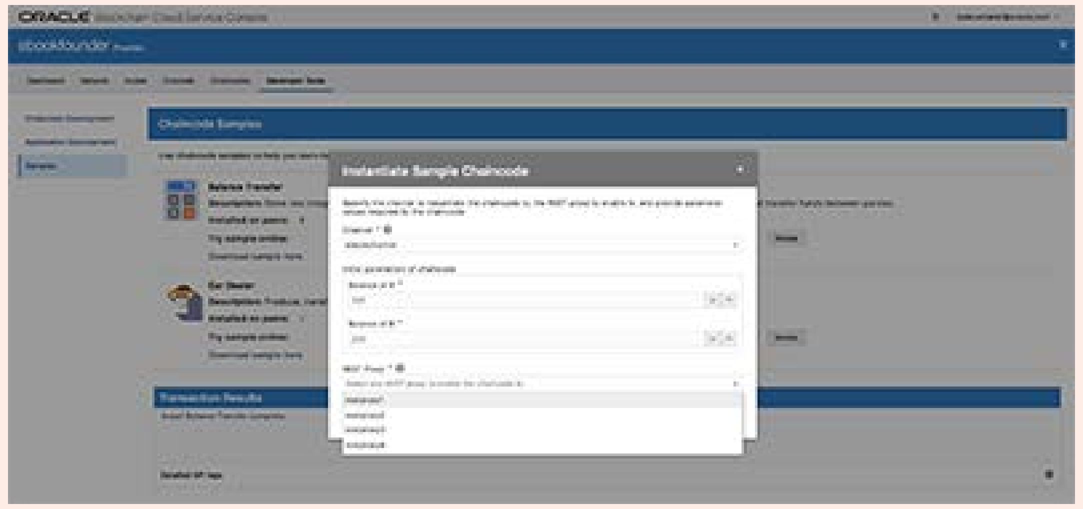
Install Sample Chaincode

Select the sample that you want to install on the peer(s) of the organization. You can install the chaincode on one or more peers.


ORACLE Blockchain Cloud Service Console
Chaincode Samples
Install Example02 Chaincode
Transaction Results

Instantiate Sample Chaincode

Select the channel to instantiate the chaincode, then Select the REST proxy server to enable & test the instantiated chaincode. Enabling instantiated chaincode on the rest proxy will assist the application developer with the API to invoke.


ORACLE Blockchain Cloud Service Console
Chaincode Samples
Instantiate Sample Chaincode
Transaction Results

Invoke and Test Sample Chaincode

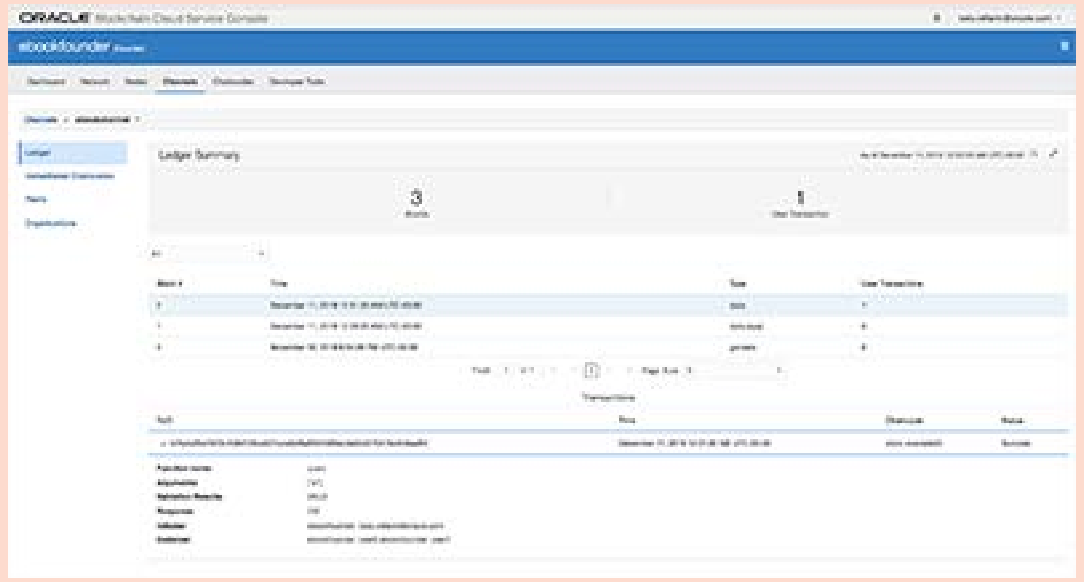
Invoke and pass the default payload defined by click on the invoke button. Transaction Results are returned with values, and the API details field displays the detailed log of all blockchain processes performed from invoking the transaction.

Free SDK →

Free Cloud Trial →

Transaction Results

Developers can go to the channels tab and click on the ledger to view the transaction invoked.


Ledger Summary

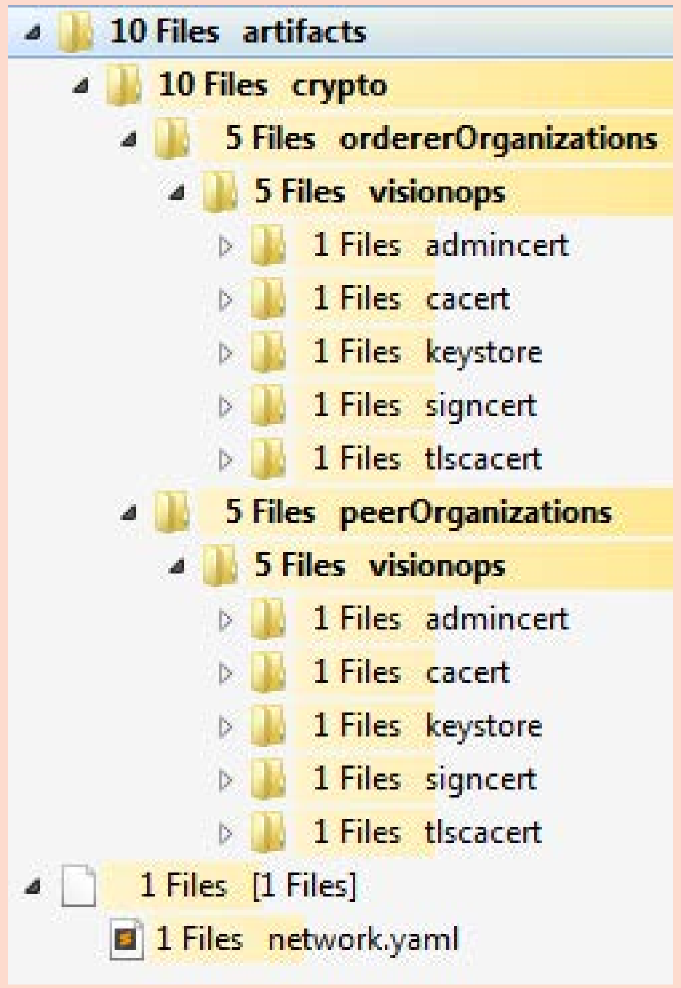
Application Development

In addition to the REST APIs provided by OBP, applications can use a software development kit (SDK) to access the APIs that permit queries and updates to the ledger. You can install and use the Hyperledger Fabric SDKs to develop applications connecting to OBP and invoking its chaincodes. This includes development SDKs to build client applications in Go (preview), Java, or Node.js, which you can download from Hyperledger Fabric site or from the OBCS Console's Developer Tools tab under Application Development page. The SDKs on this page are the same as those provided by Hyperledger.

Note: the SDKs provided with Fabric require minor adaptation to their underlying network libraries in order to work with the load balancer infrastructure in Oracle Cloud Infrastructure. Follow documentation to download the scripts to automate these adaptations or to perform them manually.

Application Development

Once downloaded, client SDKs will need connection profile info to interact with the current OBP instance, and certificates to validate signatures of fabric-ca, peers, and orderers. To simplify the configuration, OBP provides a pre-assembled developer package with connection information you can download from the Console's Developer Tools tab under Application Development page. This includes all the blockchain connection information and relevant certificates as shown below.


```
10 Files  artifacts
  10 Files  crypto
    5 Files  ordererOrganizations
      5 Files  visionops
        1 Files  admincert
        1 Files  cacert
        1 Files  keystore
        1 Files  signcert
        1 Files  tlscacert
    5 Files  peerOrganizations
      5 Files  visionops
        1 Files  admincert
        1 Files  cacert
        1 Files  keystore
        1 Files  signcert
        1 Files  tlscacert
1 Files  [1 Files]
  1 Files  network.yaml
```

Developing and Deploying Chaincodes

OBP supports Smart Contracts using application chaincodes, which are the specific programming modules that a peer can invoke on transaction request. The chaincode defines the assets and business logic for the desired transactions. Chaincodes are deployed in two steps. First they must be installed by each organization on one or more of its peers, which copies the chaincode package with the source code to the specified peers. Then they must be instantiated, which is a Hyperledger Fabric process that builds the chaincode from the source (which was copied to the peer during the install step), binds it to a channel, starts a chaincode execution container for this chaincode and initializes by calling its Init() method.

Chaincode Development - Basics

Chaincode programs written in Go, Node.js, or Java use the basic Hyperledger Fabric stub interface functions available in OBP to create, read, update, and delete records in the ledger. Note that this is not the same as adding blocks. Existing blocks are unchangeable and new blocks are generated by the ordering service and are appended by the peers. Each block contains new transactions, including their results, endorsement signatures, timestamps, and other metadata.

The ledger changes performed through the stub interface are captured by a peer into the transaction Read-Write Set (RWSet) and are considered simulated until transaction commits. Before the commit, there's no finality, and no other peer will see these changes in their copy of the ledger.

The role of the application chaincode is to evaluate the input parameters and current ledger values or historical data and produce output results, which are captured and signed by the peer as RWSet that is returned to the client.

Only after the client examines the results and sends them to the ordering service, then orderers validate and include transaction in a block sent to the peers, and peers validate and commit the transaction will the output results be committed in the ledger. At commit time the updated values are written to the world state DB, and commit flag is set for the transaction in the block that's been appended to the chain, which together with other blocks provides the transaction history log.

OBP chaincode can be written in Go, node.js, and, Java. For the examples below we use Go. The chaincode modules are required to implement two interface functions that can be invoked by a peer node:

`Init(stub ChaincodeStubInterface) pb.Response`
This is used to set up any initial state information in the ledger (e.g., starting account balance, starting inventory, etc.) – called at instantiate and upgrade time, but it can be a null function if the chaincode doesn't require an initial state.

`Invoke(stub ChaincodeStubInterface) pb.Response`
This is used to extract chaincode method being invoked and arguments passed to it, and then dispatch to the appropriate function in the chaincode.

`stub *shim.ChaincodeStubInterface` provides functions to access state and transaction history data via calls to peer as well retrieve invocation parameters, e.g.:

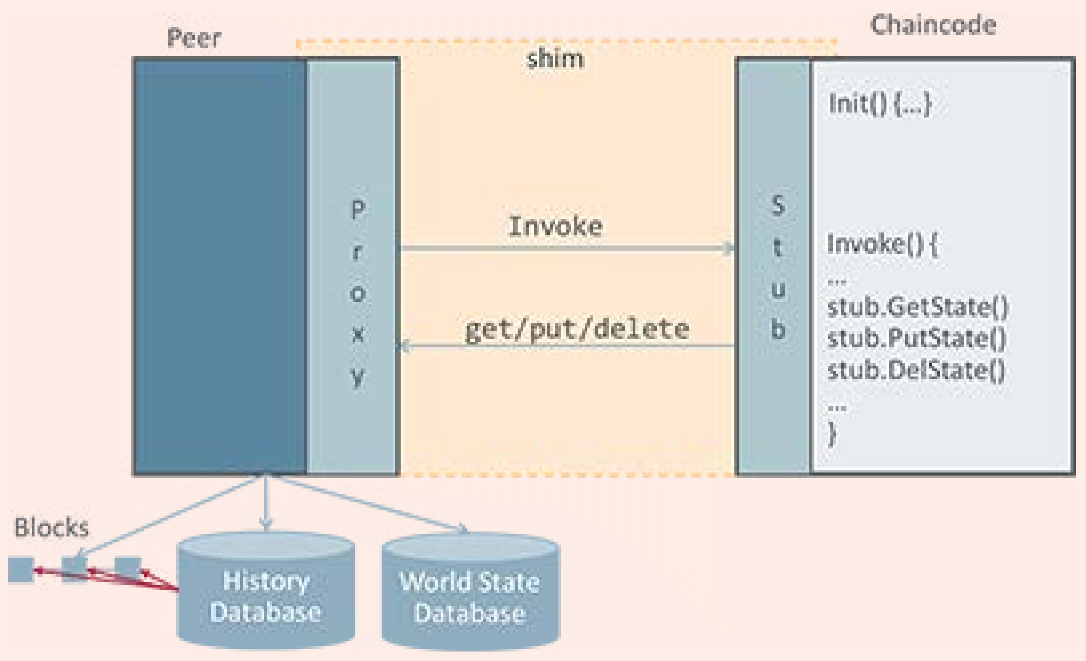
```
function, args := stub.
GetFunctionAndParameters()
```

Then using function you can dispatch the execution to a specific functional routine:

```
// Handle different functions
if function == "initLedgerA" { // set
initial state of ledger
    return t.initLedgerA(stub, args)
} else if function == "initLedgerB" { //
set initial state of ledger
    return t.initLedgerB(stub, args)
} else if function == "initVehiclePart"
{ //create a new vehiclePart
    return t.initVehiclePart(stub, args)
} else if function ==
"transferVehiclePart" { //change owner
of a specific vehicle part
    return t.transferVehiclePart(stub,
args)
}
```

Typical chaincode operates on the ledger records that represent some sort of "asset", for example, a document and its related metadata, electronic record, like an invoice or purchase order, or a digital representation of some physical assets, like cars, buildings, art works, etc. The business logic in the chaincode will generally implement CRUD (Create, Read, Update, Delete) functions on the assets, and in the process examine certain conditions and optionally trigger events.

To perform the CRUD functions, the chaincodes do not read or write ledger directly, rather they use Hyperledger Fabric shim API to invoke these operations in a peer node as shown in the diagram below.



Create

Create or Init functions add assets to the ledger. Depending on the nature of the business network, create functions could add an asset such as a new account, invoice, vehicle, real estate property, or contract. Create functions can include additional info, such as the owner, opening value, location, and other relevant attributes.

In our Car Dealer sample, when the chaincode adds a new part to the inventory ledger it uses **`PutState()`** API in the code snippet below:

```
// ==== Create vehiclePart object and
marshal to JSON ====
objectType := “vehiclePart”
//vehiclePart :=
&vehiclePart{objectType, serialNumber,
assembler, assemblyDate, name, owner}
vehiclePart := &vehiclePart{objectType,
serialNumber, assembler, assemblyDate,
name, owner, recall, recallDate}
vehiclePartJSONasBytes, err := json.
Marshal(vehiclePart)
if err != nil {
    return shim.Error(err.Error())
}
// === Save vehiclePart to state ===
err = stub.PutState(serialNumber,
vehiclePartJSONasBytes)
err != nil {
    return shim.Error(err.Error())
}
```

`serialNumber` key in the **`PutState()`** API will be included by the peer in the RWSet along with all the attribute values.

Read

Read functions are queries on the ledger. Chaincodes typically implement a variety of queries, from reading values from a specific asset to returning all assets that match certain criteria. For example, the chaincode could enable reading the value of a specific account, return all vehicles of a particular type or color, list all properties that contain certain keywords in the description, or all contracts with a particular vendor.

To read the ledger, chaincode uses GetState() API as shown in the code snippet below where we read information for a specific vehicle by referring the key **`chassisNumber`**:

Free SDK →

Free Cloud Trial →

```
vehicleAsBytes, err := stub.
GetState(chassisNumber)
    if err != nil {
        return “Failed to get vehicle:”,
err
    } else if vehicleAsBytes == nil {
        return “Vehicle does not exist”,
err
    }

    vehicleToTransfer := vehicle{}
    err = json.Unmarshal(vehicleAsBytes,
&vehicleToTransfer) //unmarshal it aka
JSON.parse()
    if err != nil {
        return “”, err
    }
```

Note that after we get **vehicleAsBytes**, we unmarshal it into a JSON structure **vehicleToTransfer**.

Update

Update functions simply change values on an already existing key or set of keys. For example, an update could deposit or withdraw funds to an account, change the condition of a vehicle, transfer ownership of a property, or modify the status of a contract. In our Car Dealer example, we transfer vehicle ownership to a new owner by adding the following code after the snippet above:

```
vehicleToTransfer.Owner = newOwner //
change the owner

vehicleJSONBytes, _ := json.
Marshal(vehicleToTransfer)
    err = stub.PutState(chassisNumber,
vehicleJSONBytes) //rewrite the vehicle
    if err != nil {
        return “”, err
    }
    return “”, nil
```

We change the Owner attribute to the value of newOwner, masrhal the vehicleToTransfer structure again and use **PutState()** to save updated values for the **chassisNumber** key.

Delete

The delete function doesn’t really delete the data, but just marks the key as “deleted”, indicating that the specified asset is no longer available. This could represent the closing of an account, selling a product out of inventory, or completing a contract. In the Car Dealer chaincode, we have the following example using **DelState()** API on **serialNumber** key:

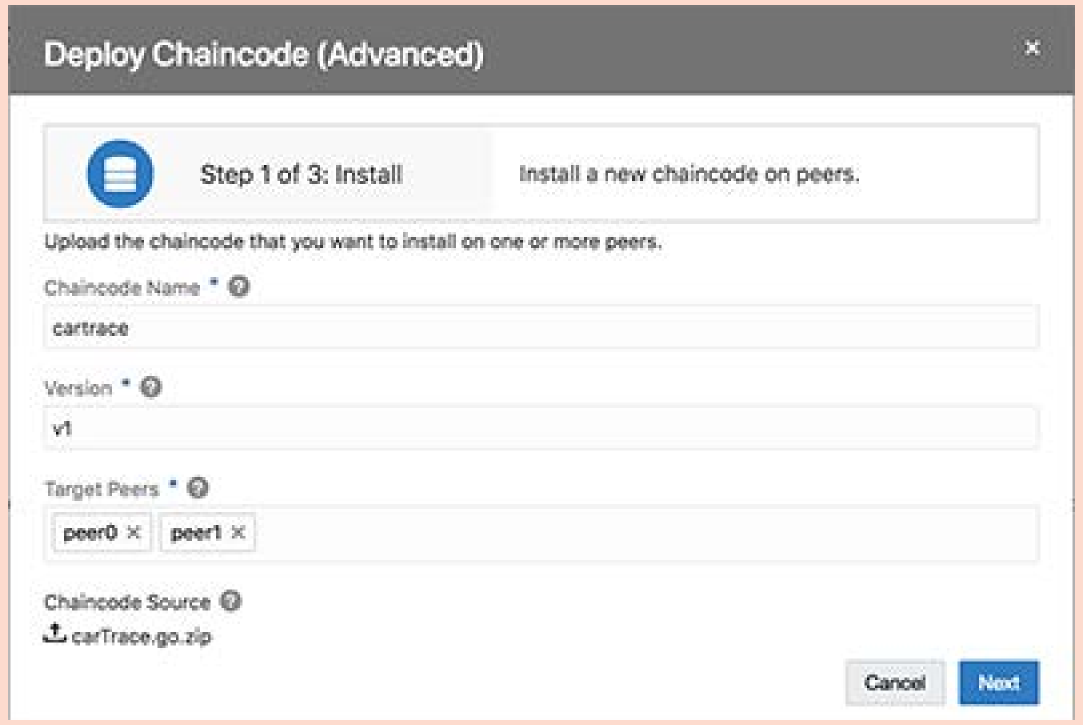
```
err = stub.DelState(serialNumber) //
remove the vehiclePart from chaincode
state
if err != nil {
            return shim.Error(“Failed to
delete state:” + err.Error())
}
```

Compiling Chaincode

Chaincode is a program, written in Go, node.js, or Java that implements a prescribed interface, and therefore it needs to be compiled before you run it. When deploying it on OBP, the compilation is automatically taken care of during deployment. But it's a good idea to compile it locally first to ensure there are no compilation errors before installing the chaincode in OBP. Using a local compile environment you can quickly check for any errors since it's much easier to check the compiler messages and make corrections in the local environment. For local compiles you need to download all the necessary packages of the chaincode's programming language and **Hyperledger Fabric**. Every chaincode program implements chaincode shim API. As a good practice, you chaincodes should be compiled locally to ensure there are no compilation errors before installing them in OBP. Developers can setup their development environment and start developing chaincode by following the steps mentioned in Hyperledger Fabric documentation.

Installing & Instantiating Chaincode

Chaincodes are installed on specific peers and then instantiated on specific channels, giving you complete control over which transaction types are visible and available to which peers on any channel. In this example, the name of the chaincode is **cartrace** and its version is **v1**.


Deploy Chaincode (Advanced)

Step 1 of 3: Install — Install a new chaincode on peers.

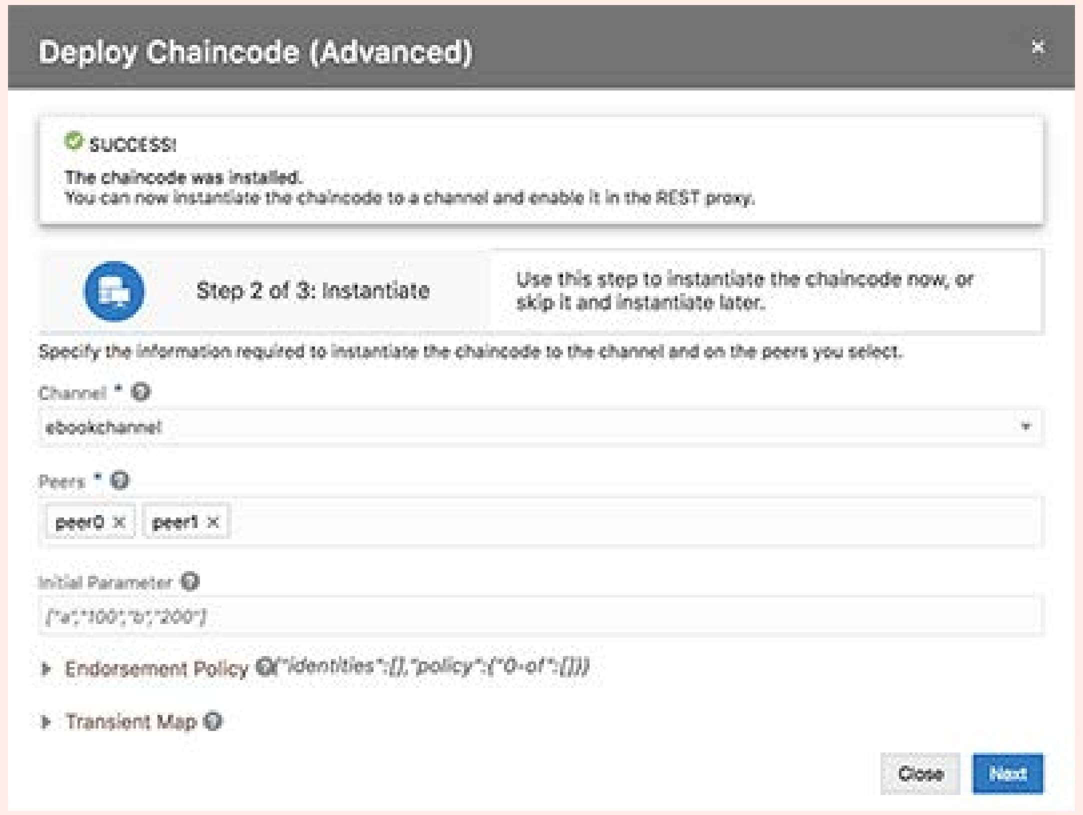
Upload the chaincode that you want to install on one or more peers.

Chaincode Name *: cartrace

Version *: v1

Target Peers *: peer0, peer1

Chaincode Source: carTrace.go.zip


Deploy Chaincode (Advanced)

SUCCESS!
The chaincode was installed.
You can now instantiate the chaincode to a channel and enable it in the REST proxy.

Step 2 of 3: Instantiate — Use this step to instantiate the chaincode now, or skip it and instantiate later.

Specify the information required to instantiate the chaincode to the channel and on the peers you select.

Channel *: ebookchannel

Peers *: peer0, peer1

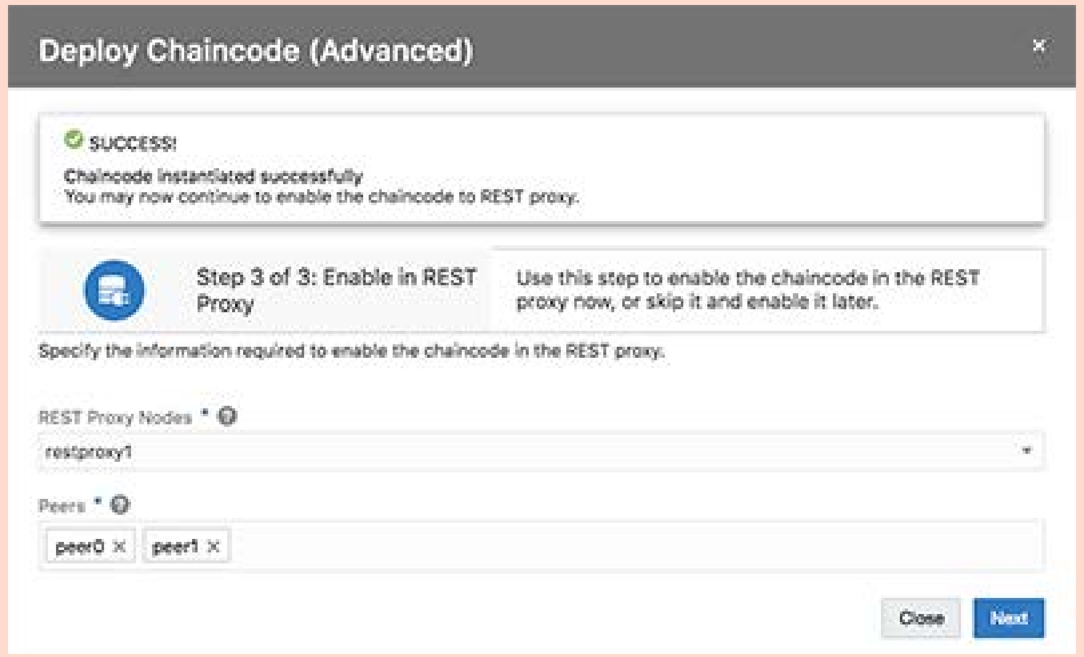
Initial Parameter: ["a","100","b","200"]

Endorsement Policy {"identities":[],"policy":{"0-of":[]}}

Transient Map

After installing the chaincode on the founder organization's nodes, you need to decide which channels to expose it on. Each channel can have multiple chaincodes, and each chaincode can be instantiated on multiple channels depending on the needs of your blockchain network.

Finally, you can enable access to chaincode via REST proxies.



Deploy Chaincode (Advanced)

SUCCESS!
Chaincode instantiated successfully
You may now continue to enable the chaincode to REST proxy.

Step 3 of 3: Enable in REST Proxy — Use this step to enable the chaincode in the REST proxy now, or skip it and enable it later.

Specify the information required to enable the chaincode in the REST proxy.

REST Proxy Nodes *: restproxy1

Peers *: peer0, peer1

In a blockchain network with multiple instances and organizations, each organization's blockchain instance administrator must install the chaincode on their local peers. However, once chaincode has been instantiated for a particular channel, when the same chaincode is installed on any peer on that channel, the instantiation is automatically extended to that peer – that is, the chaincode is compiled and loaded into a chaincode execution container linked to that peer. No explicit instantiation is required in this case.

However, if an organization wants to have the chaincode accessible via their own REST proxy, they need to edit the proxie's configuration (Edit Configuration option on the proxie's Actions menu in the Nodes tab) and select the appropriate channel, chaincode, and peer nodes to expose as REST end point.

Implementing Custom Chaincode Functions

Let's take a simple use case and explore its custom chaincode functions.

Use Case

In this use case, a large automaker and its dealers have created a blockchain network to streamline their inventory management activities. Blockchain helps them reduce the time required to reconcile shared inventory information with the vehicle and parts audit trail.

The sample includes a chaincode to manage the production, transfer, and querying of vehicle parts; the vehicles assembled from these parts; and transfer of the vehicles.

To locate and download the sample, please visit https://cloud.oracle.com/blockchain/additional-resources.

The cartrace chaincode includes some custom methods for managing manufacturer-to-dealer transactions:

	Methods	Arguments & Key Code

Vehicles	**initVehicle**
	Add a new vehicle to the ledger
	
	This creates a new vehicle with the attributes provided by the arguments, e.g., "mer1000001", "mercedes", "c class", "1502688979", "ser1234", "mercedes", "false", "1502688979"
		Chassis ID number, Make, Model, Assembly timestamp, Owner, Recall status, Recall timestamp
		
		Sample code to add a new vehicle to the ledger:
		`vehicle := &vehicle{objectType, chassisNumber, manufacturer, model, assemblyDate, airbagSerialNumber, owner, recall, recallDate}`
		`vehicleJSONasBytes, err := json.Marshal (vehicle)`

	readVehicle
	Read the status of a vehicle
	
	This retrieves the vehicle status based on the chassis number in the arguments, e.g.,: “mer1000001”
		Chassis ID number
		
		Sample Code to retrieve the vehicle info from the ledger:
		`chassisNumber = args[0]`
		`valAsbytes, err := stub.GetState(chassisNumber)`

	transferVehicle
	Change the ownership value of vehicle
	
	This transfers the vehicle from one owner to another based on the arguments, e.g.,:
	Sample arguments are:
	“mer1000001”,
	“SamDealer”,
	“JudeDealer”
		Chassis ID number, Current owner, New owner
		
		Sample Code:
		`chassisNumber := args[0]`
		`currentOwner := strings.ToLower(args[1])`
		`newOwner := strings.ToLower(args[2])`
		
		Get the vehicle based on the chassis Number and update the vehicle info
		`vehicleAsBytes, err := stub.GetState(chassisNumber)`
		
		`vehicleJSONBytes, _ := json.Marshal(vehicleAsBytes)`
		`err = stub.PutState(chassisNumber, vehicleJSONBytes)`

Free SDK →   Free Cloud Trial →

Vehicle Parts

Methods	Arguments & Key Code
initVehiclePart Add a new part to the ledger. This adds a new part to the vehicle	Part ID, Assembler, Assembly timestamp, Owner, Recall status, Recall timestamp Sample Code similar to **initiVehicle**
readVehiclePart This queries on part ID and returns vehicle info related to the part	Part ID Sample Code similar to **readVehicle**
transferVehiclePart This transfers the vehicle part from one vehicle to other	Part ID, Current owner, New owner Sample Code similar to **readVehicle**

getHistoryForRecord
This returns all the transactions for a given key of a record
	RecordID
	
	Sample Code to retrieve the history of the transaction from the ledger
	
	`recordKey := args[0]`
	`resultsIterator, err := stub.GetHistoryForKey(recordKey)`

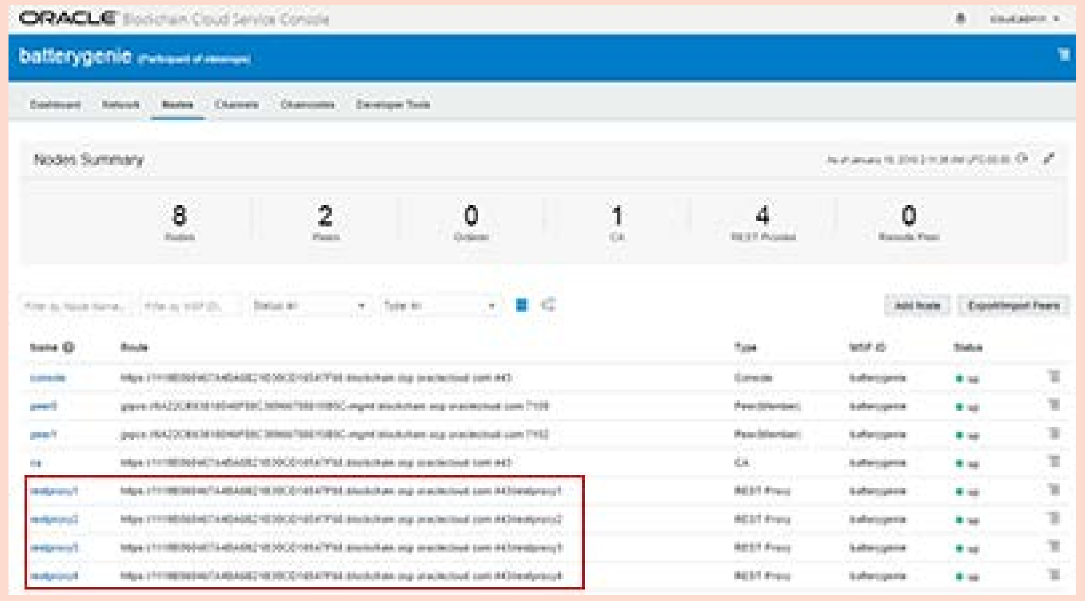
getVehiclePartByRange
This performs a range query based on the start and end keys provided
	PartID, StartDate, EndDate
	
	Sample Code to retrieve the history of the transaction from the ledger
	
	`startKey := args[0]`
	`endKey := args[1]`
	`resultsIterator, err := stub.GetStateByRange(startKey,`
	`endKey)`

Invoking Chaincodes via REST API

OBP includes a REST proxy to enable chaincode functions to be accessed via RESTful APIs. The APIs are configured in the REST proxy when chaincodes are deployed using OBCS wizards, or can be added independently using Edit Configuration menu pulldown from the REST proxy Actions menu on the Nodes tab. See documentation for full REST API details. The API functions described below are provided by the REST proxy for querying gateway versions, invoking transactions and queries, and subscribing to events. In addition, OBCS provides a comprehensive set of REST APIs for configuration and monitoring that handle the administration functions similar to those in the Console web UI.

Each REST API requires an endpoint, header, and credentials for authentication. Most also require a JSON body. The REST endpoints contain two components in the form of **`https://{obcsRestURL}/{resource-path}`**. The **`obcsRestURL`** is in the form **`<rest_server_url:port/restproxy#>`**, which you can get from the Nodes tab in the Console.

Find the row for one of the REST proxies and copy the contents of the Route column as shown below:



Resource-path depends on the specific API and is shown below. Note: to distinguish the application REST API endpoints from the administration ones, the resource-paths below start with **/bcsgw/rest**, while the administration and statistics APIs start with **/console/admin/api**. Following sections describe application operations APIs for querying the ledger, invoking transactions, and subscribing to events. For administration and statistics APIs see the relevant Task categories in the navigation bar in the online documentation. To use a visual API explorer to test drive these APIs, visit Oracle Blockchain Platform API Catalog and select either Console or Gateway API spec in the dropdown.

Querying REST Proxy Version

View Version

Check connectivity to the REST proxy (gateway) and verify its version number.

Endpoint: **`https://{obcsRestURL}/bcsgw/rest/version`**

Returns: gateway's version

Transactions and Queries

Invoke a Query

Execute a chaincode function that returns information, without committing the transaction.

Endpoint: **`https://{obcsRestURL}/bcsgw/rest/v1/transaction/query`**

Returns: response payload from the query function and its encoding

Invoke a Method (Synchronous)

Execute a chaincode function and commit the transaction to the ledger.

Endpoint: **`https://{obcsRestURL}/bcsgw/rest/v1/transaction/invocation`**

Returns: response payload from the function and transaction ID

Invoke a Method (Asynchronous)

Execute a chaincode function in asynchronous mode and commit transaction to the ledger.

Endpoint: `https://{obcsRestURL}/bcsgw/rest/v1/transaction/asyncInvocation`

Returns: transaction ID created for this transaction

View the Status of a Specified Transaction

Check the status of a previously submitted transaction and retrieve the response payload.

Endpoint: `https://{obcsRestURL}/bcsgw/rest/v1/transaction`

Returns: transaction status and response payload from chaincode function

GetTransactionID

Request an asynchronous mode transaction ID by channel name for a transaction you will invoke later in asynchronous mode.

Endpoint: `https://{obcsRestURL}/bcsgw/rest/v1/transaction/getTxID`

Returns: transaction ID and nonce

Events

Subscribe

Register a subscription to an event, specifying an event type, event name, channel, callback URL, and expiration time.

Endpoint: `https://{obcsRestURL}/bcsgw/rest/v1/event/subscribe`

Returns: subscription ID

Unsubscribe

Remove subscription registration for one or more events using subscription ID.

Endpoint: `https://{obcsRestURL}/bcsgw/rest/v1/event/unsubscribe`

Returns: subscription ID and status

To test REST APIs before using them from applications, you can use command line tool curl or GUI tools, such as Postman.

In a command line these REST calls look like this:

```
curl -i -u <user>:<pwd> -H Content-
type:application/json -X POST -d  @
body.json https://<rest_server_
url:port>/<restproxy#>/bcsgw/rest/v1/
transaction
```

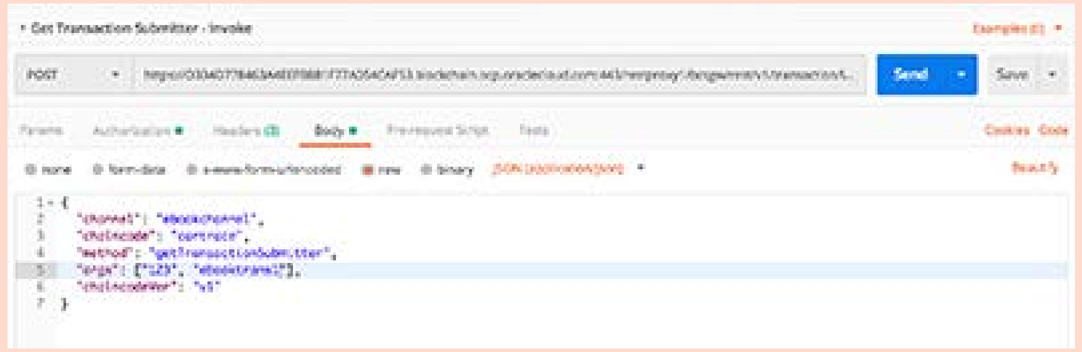
where parameters inside <> need to be replaced with actual values and body.json referenced by –d parameter is a text file that needs to contain the JSON body with the details of the request, e.g.:

```
{
“channel”:  “dealernet”,
“chaincode”:  “end2end”,
“method”:  “myfunc”,
“args”:  [“a”,”b”,”3”]
}
```

In this JSON body, we are requesting invocation of end2end chaincode on the channel dealernet and passing to the chaincode method myfunc and an array of arguments.

Inside the chaincode an Invoke dispatcher function will retrieve the method name and arguments and dispatch the call to the requested method.

If you prefer to use a visual API test tool, similar request using Postman would look like this:

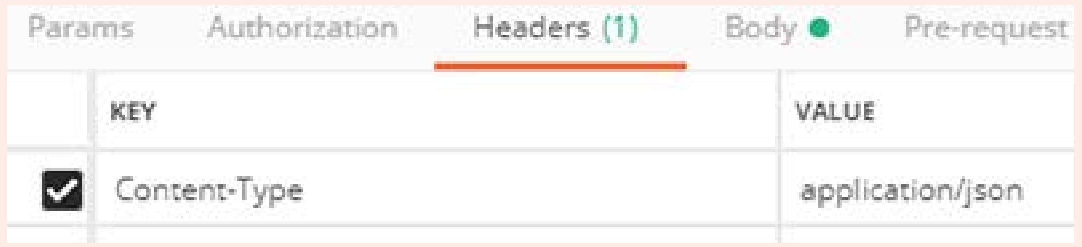


```
{
    "channel": "ebookchannel",
    "chaincode": "cartrace",
    "method": "getTransactionSubmitter",
    "args": ["123", "ebooktrans1"],
    "chaincodeVer": "v1"
}
```

In this UI the left pane contains a Collection – a library of pre-defined APIs. The main pane shows POST request type in the grey bar and next to it the REST endpoint with {{hostname}}:{{port}} indicating these are variables pre-set for this OBCS Workshop configuration to point to my OBCS Rest proxy instance. Below the grey bar is a series of tabs:

In the Authorization tab you must specify your credentials for access to the REST proxy. These could be the administrator's credentials of the OBCS accounts (the same ones you use to login to the Console), or different credentials that have been added to the IDCS app of this OBCS instance and configured for the **`RESTPROXY<N>_User`** role. Refer to Set Up Users and Access Roles in the online documentation for detailed steps

The Header tab must specify Content-Type header with application/json value as shown below.



	KEY	VALUE
✓	Content-Type	application/json

And the Body tab must contain the JSON body with details of the request, which is the same as the contents of the body.json file used in the curl example above.

When everything's set, click the **Send** button to send your request and the response will be shown below.

Initializing the Ledger

Now, let's start using the API's to trigger the chaincode functions.

Before you can start trading assets or executing other business transactions, you may need to establish the initial state of the ledger. For example, setting up your account balances, initial inventory, or open contracts.

Continuing the car dealer example, we will initialize the ledger with a variety of vehicles and parts. The channel is **ebookchannel**, the chaincode is still **cartrace**, the custom function is **initVehicle**, and the arguments include the chassis ID number, make, model, assembly timestamp, airbag serial number, owner, recall status, and recall timestamp.

Free SDK →   Free Cloud Trial →

In REST format this looks like:

```
curl -u <user>:<pwd> -H Content-
type:application/json -X POST -d @
body.json  https://<rest_server_
url:port>/<restproxy#>/bcsgw/rest/v1/
transaction/invocation
```

The JSON body in body.json file:

```
{
"channel": "ebookchannel",
"chaincode": "cartrace",
"method": "initVehicle",
"args": ["porsche1000001", "porsche",
"cayenne", "1541542934", "ser1234",
"detroit-autos", "false", "1541542934"]
```

The response is a simple success message, with the resulting transaction ID.

```
{
"returnCode": "Success",
"txid": "9d0489543542e2ef53cc155dbdb52
31d5770640b15afc0b236a3089f2ed0835c"
}
```

To bulk load multiple vehicles and vehicle parts in the ledger, you can put the curl commands into a script and use multiple body.json files or inline json using a single quoted string after curl's –d flag instead of a file reference like this:

```
curl -u $USER:$PASSWORD -H "Content-
type:application/json" -X POST -d
'{"channel":"'"ebookchannel"'",
"chaincode":"'cartrace'","method":
"initVehiclePart",
"args":["abg1234", "panama-parts",
"1502688979", "airbag 2020",
"'"$MANU_NAME"'", "false",
"1502688979"]}'
https://<rest_server_
url:port>/<restproxy#>/bcsgw/rest/v1/
transaction/invocation
```

Invoking Operations

Once the ledger is initialized with some assets, the participants are ready to execute authorized transactions. The first step would be querying a vehicle chassis ID to see if it is available using the query API (resource-path for queries is /bcsgw/rest/v1/transaction/query in the endpoint):

```
{
"channel": "ebookchannel",
"chaincode": "cartrace",
"method": "readVehicle",
"args": ["porsche1000001"]
}
```

The response comes back with all of the details from the asset's ledger entry (edited for readability):

```
{
"returnCode": "Success",
"result": {
"payload": "{
"docType":"vehicle",
"chassisNumber": porsche1000001",
"manufacturer":"porsche",
"model":" "cayenne",
"assemblyDate":1541542934, (Date in
Epoch format)
"airbagSerialNumber":"ser1234",
"owner":"detroit-autos",
"recall":false,
"recallDate":1541542934}",
"encode": "UTF-8"
},
"txid": "b56c45db0dd61a4d7a3421bd44
251891c9ee312429df918c32187bd20aa20176"
}
```

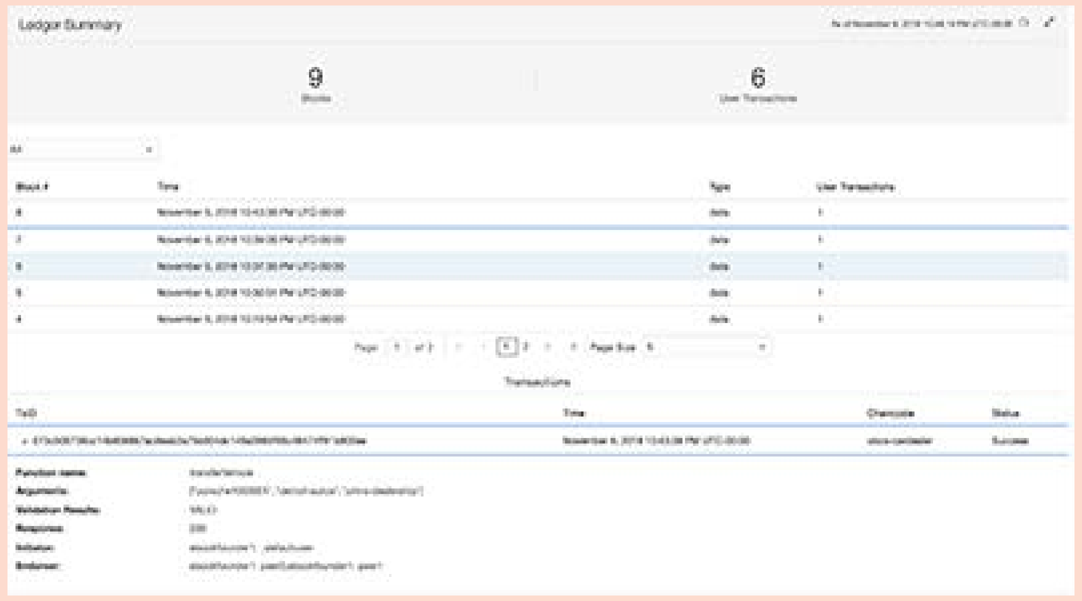
Detroit Autos then transfers the vehicle's ownership from the manufacturer to the dealership:

```
{
"channel": "ebookchannel",
"chaincode": "cartrace ",
"method": "transferVehicle",
"args": ["porsche1000001", "detroit-
autos", "johns-dealership"]
}
```

The response is again a simple success message and a new transaction ID:

```
{
"returnCode": "Success",
"txid":
"073c500739cc14b806867ac8eab3a75b9
04dc149a0965f99c48474ff91b800ee"
}
```

On the Console's Channels tab click to select ebookchannel and you can see in the ledger browser the blocks that have been created with these transactions.



Ledger Summary

9

6

[illegible]

The Get History Record for that chassis number now gives a compound result, linking the two ledger entries:

Free SDK →

Free Cloud Trial →

```
{
  "returnCode": "Success",
  "result": {
    "payload": [
      {
        "TxId":
"9d0489543542e2ef53cc155dbdb5231d5
770640b15afc0b236a3089f2ed0835c",
        "Value": {
          "docType": "vehicle",
          "chassisNumber":
"porsche1000001",
          "manufacturer": "porsche",
          "model": "cayenne",
          "assemblyDate": 1541542934,
          "airbagSerialNumber":
"ser1234",
          "owner": "detroit-autos",
          "recall": false,
          "recallDate": 1541542934
        },
      "Timestamp": "2018-11-06
22:37:36.205 +0000 UTC",
```

```
"IsDelete": "false"
      },
      {
        "TxId":
"073c500739cc14b806867ac8eab3a75b9
04dc149a0965f99c48474ff91b800ee",
        "Value": {
          "docType": "vehicle",
          "chassisNumber":
"porsche1000001",
          "manufacturer": "porsche",
          "model": "cayenne",
          "assemblyDate": 1541542934,
          "airbagSerialNumber":
"ser1234",
          "owner": "johns-dealership",
          "recall": false,
          "recallDate": 1541542934
        },
       "Timestamp": "2018-11-06
22:43:38.548 +0000 UTC",
        "IsDelete": "false"
      }
```

```
],
    "encode": "UTF-8"
  },
  "txid":
"5409c942d8c7c46466d7693dcdab2313c9
8e220dc781fbff77c3abac645ef319"
}
```

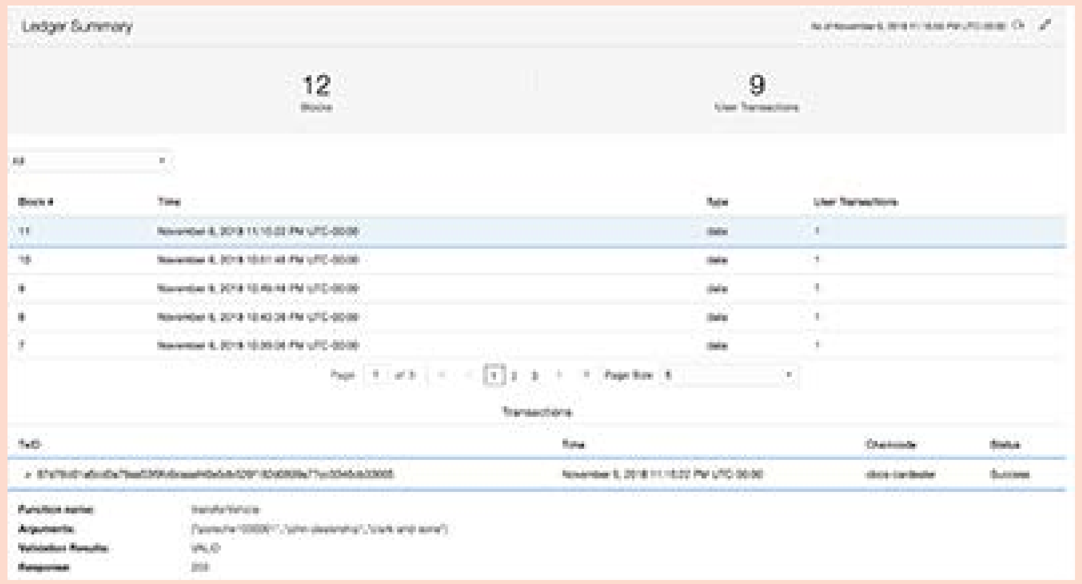
Finally, when the dealer sells the car they transfer the ownership again, which writes a new block to the ledger:

```
{
"channel": "ebookchannel",
"chaincode": "cartrace ",
"method": "transferVehicle",
"args": ["porsche1000001", "john-
dealership", "clark-and-sons"]
```

This action returns a new transaction ID:

```
{
"returnCode": "Success",
"txid": "87d78d01a6cd0a79aa53f9f
c0caaaf40e5db529f182d0899a77cc
3346cb33665"
}
```

The ledger browser in the Console shows that this latest transaction has completed:



Ledger Summary

12 Blocks

9 User Transactions

Block #	Time	Type	User Transactions
11	November 6, 2018 11:10:22 PM UTC-00:00	data	1
10	November 6, 2018 10:51:48 PM UTC-00:00	data	1
9	November 6, 2018 10:49:44 PM UTC-00:00	data	1
8	November 6, 2018 10:43:26 PM UTC-00:00	data	1
7	November 6, 2018 10:35:06 PM UTC-00:00	data	1

Page 1 of 3 | 1 2 3 | Page Size 5

Transactions

TxID	Time	Chaincode	Status
87d78d01a6cd0a79aa53f9f[illegible]	November 6, 2018 11:10:22 PM UTC-00:00	[illegible]	Success

Function name: transferVehicle
Arguments: [illegible]
Validation Results: VALID
Response: 200

Additional detail for event subscription and callbacks are shown in Events – Publish & Subscribe section under Advanced Topics chapter.

Full details for the REST API Endpoints are available on the documentation site at https://docs.oracle.com/en/cloud/paas/blockchain-cloud/rest-api/rest-endpoints.html

Advanced Topics

ABAC – Attribute Based Access Control

Developers can utilize Hyperledger Fabric client identity chaincode library (cid) to make authorization decisions based on an attribute value and/or the MSPID (Memership Service Provider Id) associated with the client. This library provides APIs that allow chaincode to retrieve the MSP ID used to issue the certificate of the invoker and all the attributes associated with the certificate provided when it was being issued via SDK's register() and enroll() API calls to fabric-ca. Attributes are key value pairs like `email=test@oracle.com`. For example, this can be used to allow or disallow a specific operation in the chaincode like search based on the attribute and value of the MSP.

The chaincode library provides functions such as:

- GetID() (string, error)
- GetMSPID() (string, error)
- GetAttributeValue(attrName string) (value string, found bool, err error)
- GetX509Certificate() (*x509.Certificate, error)
- AssertAttributeValue(attrName, attrValue string, error)

Reference: (https://github.com/hyperledger/fabric/blob/release-1.3/core/chaincode/lib/cid/interfaces.go)

Steps to setup CID module in chaincode development environment:

1. Download Golang protobuf and errors packages from golang github to your local environment
   a. https://github.com/golang/protobuf
   b. https://github.com/pkg/errors

2. Download the cid modules and dependencies from hyperledger github to your local environment from

   a. Download cid.go, interfaces.go from https://github.com/hyperledger/fabric/tree/release-1.3/core/chaincode/lib/cid

   b. Download attrgr.go from https://github.com/hyperledger/fabric/tree/release-1.4/common/attrmgr

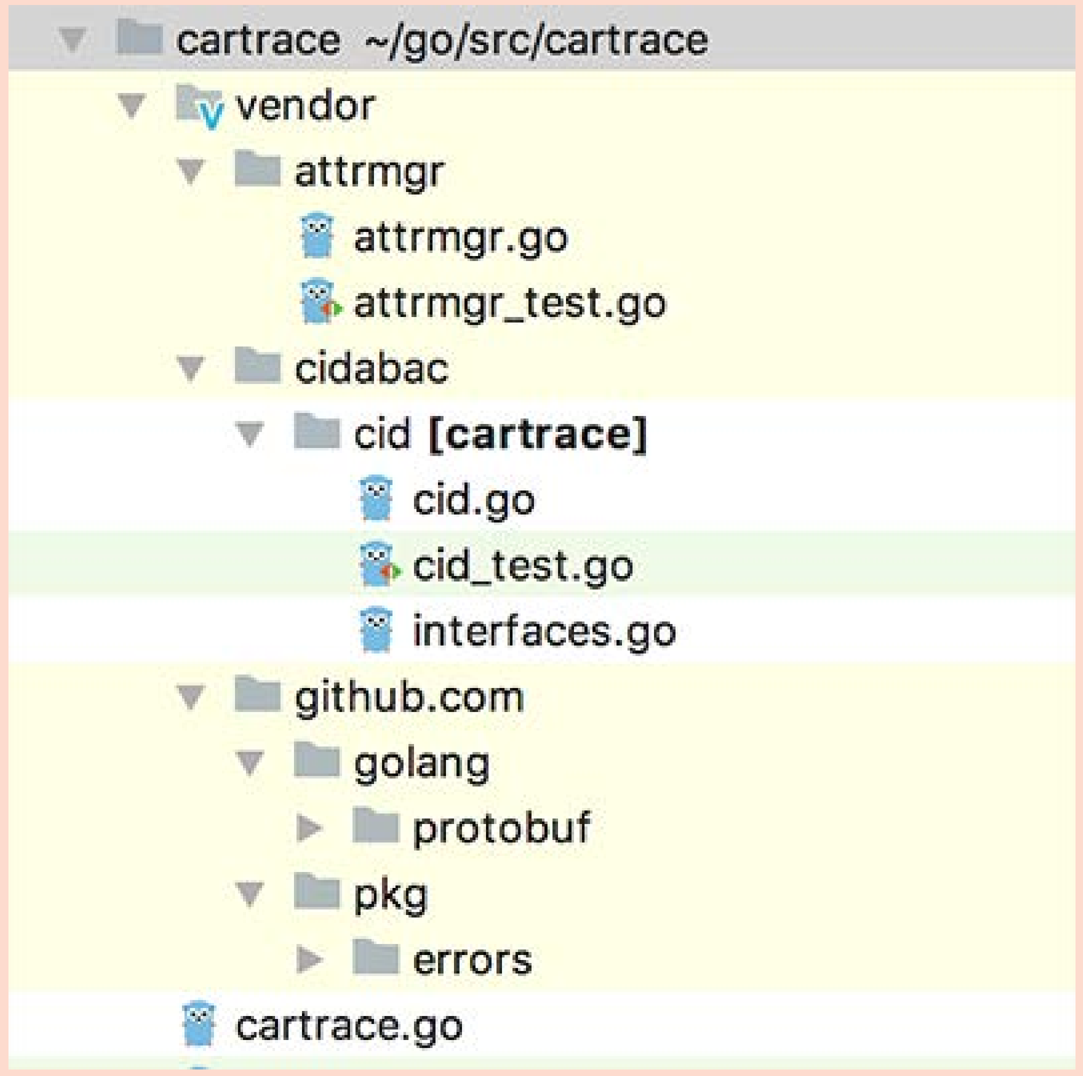
3. Setup your chaincode development environment based on go vendor dependency model as shown below.

   a. Create a vendor directory under your go project.

   b.Under the vendor directory unzip the downloaded packages from step 1 (github.com/golang/protobuf and gitbub.com/pkg/errors) as shown in the screen shot below.

   c. Under the vendor directory unzip the cid & attrmgr packages as shown in the screenshot below.

   d. cid & attrmgr module refers protobuf packages.



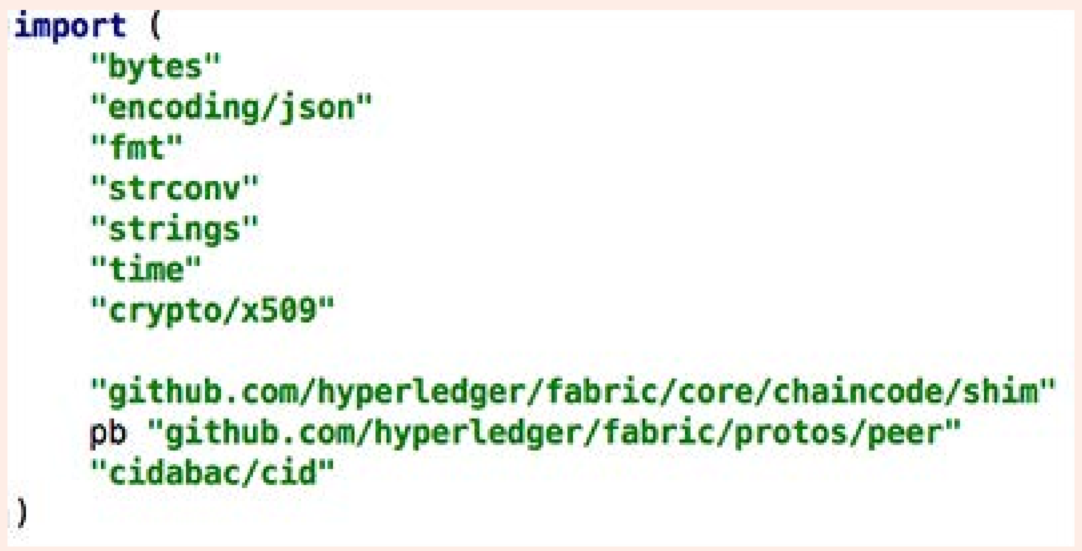
```
▼ cartrace  ~/go/src/cartrace
  ▼ vendor
    ▼ attrmgr
        attrmgr.go
        attrmgr_test.go
    ▼ cidabac
      ▼ cid [cartrace]
          cid.go
          cid_test.go
          interfaces.go
    ▼ github.com
      ▼ golang
        ▶ protobuf
      ▼ pkg
        ▶ errors
    cartrace.go
```

4. Package (zip) the root (parent) folder into cartrace.zip and deploy the chaincode in OBP using Quick or Advanced deploy wizards on the Chaincode tab.

Now, let's see how a developer can use the library defined above and create sample function in the chaincode:

In your chaincode program, import the necessary packages

- **crypto/x509**
- **shim** – github.com/hyperledger/fabric/core/chaincode/shim
- **cid** – using **cidabac/cid** path to refer to the relative resource path of **cid** under vendor directory as shown below

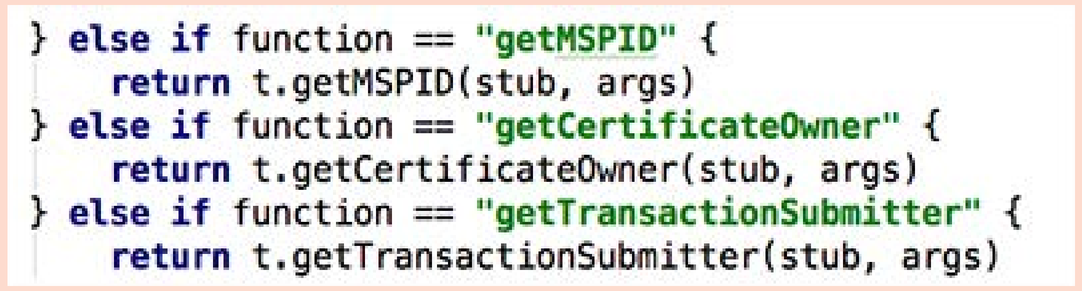


```
import (
    "bytes"
    "encoding/json"
    "fmt"
    "strconv"
    "strings"
    "time"
    "crypto/x509"

    "github.com/hyperledger/fabric/core/chaincode/shim"
    pb "github.com/hyperledger/fabric/protos/peer"
    "cidabac/cid"
)
```

Free SDK →

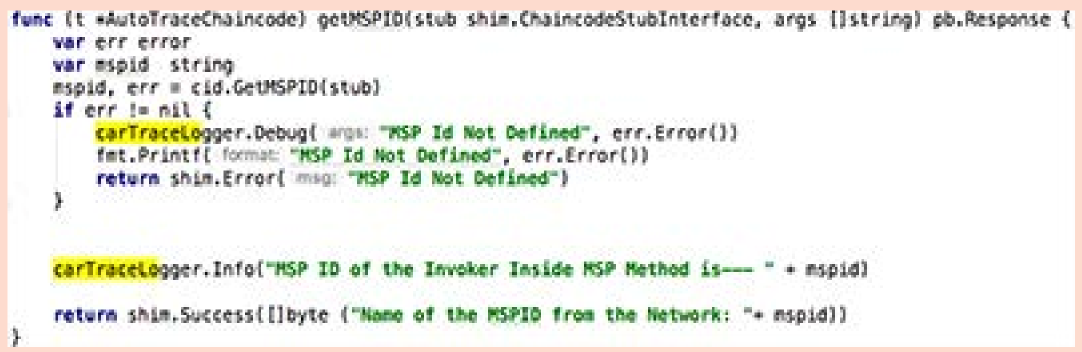
Free Cloud Trial →

Now you can write functions to retrieve Identity in your code as shown below, using cid library to retrieve the identity of the Invoker in their implementation as shown below.



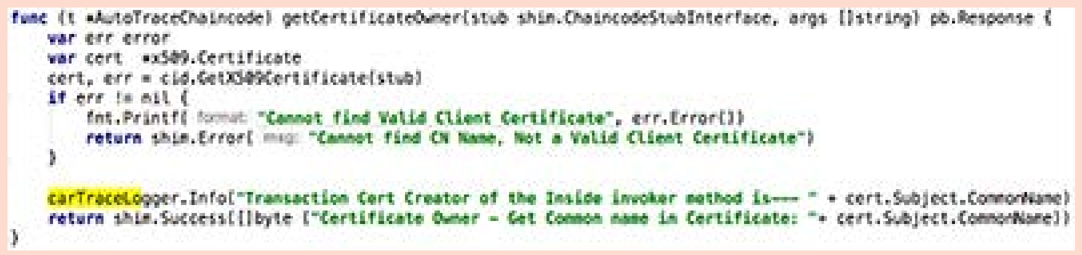
```
} else if function == "getMSPID" {
    return t.getMSPID(stub, args)
} else if function == "getCertificateOwner" {
    return t.getCertificateOwner(stub, args)
} else if function == "getTransactionSubmitter" {
    return t.getTransactionSubmitter(stub, args)
```

getMSPID



```
func (t *AutoTraceChaincode) getMSPID(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    var err error
    var mspid  string
    mspid, err = cid.GetMSPID(stub)
    if err != nil {
        carTraceLogger.Debug( args: "MSP Id Not Defined", err.Error())
        fmt.Printf( format: "MSP Id Not Defined", err.Error())
        return shim.Error( msg: "MSP Id Not Defined")
    }


    carTraceLogger.Info("MSP ID of the Invoker Inside MSP Method is--- " + mspid)

    return shim.Success([]byte ("Name of the MSPID from the Network: "+ mspid))
}
```

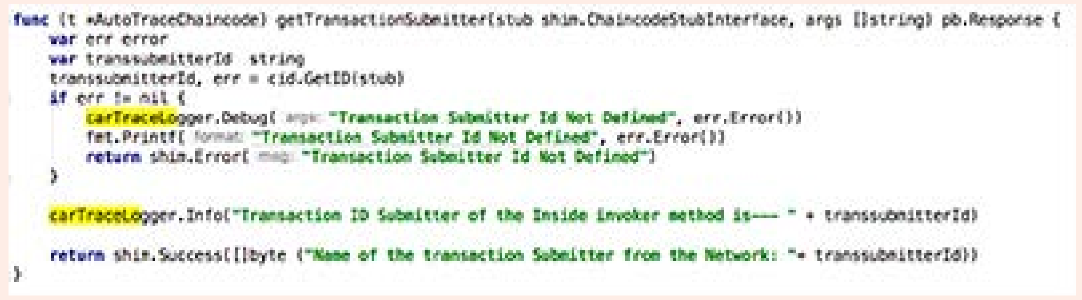
getCertificateOwner



```
func (t *AutoTraceChaincode) getCertificateOwner(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    var err error
    var cert  *x509.Certificate
    cert, err = cid.GetX509Certificate(stub)
    if err != nil {
        fmt.Printf( format: "Cannot find Valid Client Certificate", err.Error())
        return shim.Error( msg: "Cannot find CN Name, Not a Valid Client Certificate")
    }

    carTraceLogger.Info("Transaction Cert Creator of the Inside invoker method is--- " + cert.Subject.CommonName)
    return shim.Success([]byte ("Certificate Owner - Get Common name in Certificate: "+ cert.Subject.CommonName))
}
```

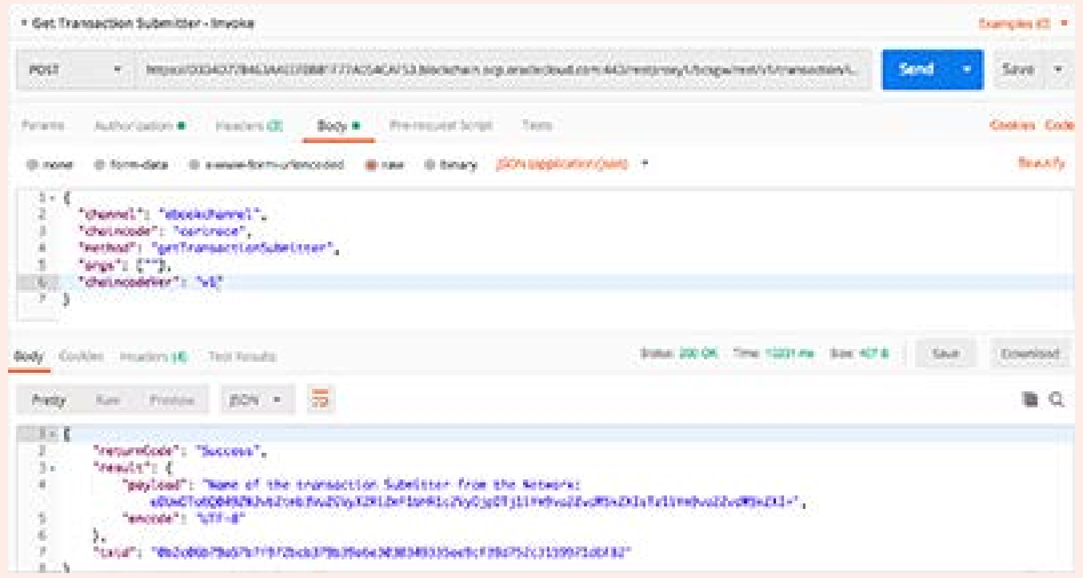
In REST format this looks like:

getTransactionSubmitter



```
func (t *AutoTraceChaincode) getTransactionSubmitter(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    var err error
    var transsubmitterId  string
    transsubmitterId, err = cid.GetID(stub)
    if err != nil {
        carTraceLogger.Debug( args: "Transaction Submitter Id Not Defined", err.Error())
        fmt.Printf( format: "Transaction Submitter Id Not Defined", err.Error())
        return shim.Error( msg: "Transaction Submitter Id Not Defined")
    }

    carTraceLogger.Info("Transaction ID Submitter of the Inside invoker method is--- " + transsubmitterId)

    return shim.Success([]byte ("Name of the transaction Submitter from the Network: "+ transsubmitterId))
}
```

While you might use these functions internally as part of your business logic, you can test the Identity Functions from Postman client using the REST API. For example, for GetTransactionSubmitter method, the REST call from Postman and the expected response are shown below.



POST https://[illegible].blockchain.ocp.oraclecloud.com:443/restproxy1/bcsgw/rest/v1/transaction/i[illegible]

```
{
    "channel": "abookchannel",
    "chaincode": "carcrace",
    "method": "getTransactionSubmitter",
    "args": [""],
    "chaincodeVer": "v1"
}
```

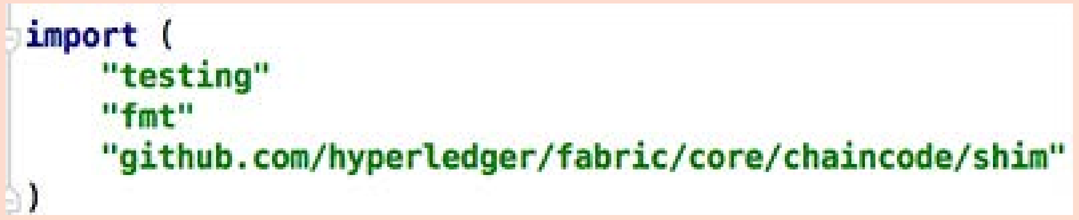
Status: 200 OK   Time: 1231 ms   Size: 427 B

```
{
    "returnCode": "Success",
    "result": {
        "payload": "Name of the transaction Submitter from the Network:
            [illegible]",
        "encode": "UTF-8"
    },
    "txid": "[illegible]"
}
```

Testing Chaincode Using Mockshim

To speed up the development and testing lifecycle of the chaincode and enable local debugging, you may want to run some unit testing locally. This is possible using a mock version of the **stub shim. ChaincodeStubInterface**, which provides a local mechanism to simulate responses for **GetState/ PutState/DelState** ledger access functions without connecting to a peer node. It enables a test module for unit testing the basic functionality of your chaincode before deploying it to Oracle Blockchain Platform. You can also use this library to build unit tests for your chaincode.

Import package "**github.com/hyperledger/ fabric/core/chaincode/shim**" and use **shim. NewMockStub** to create smart contract and test your procedures with debugging and without having to connect to a blockchain network.

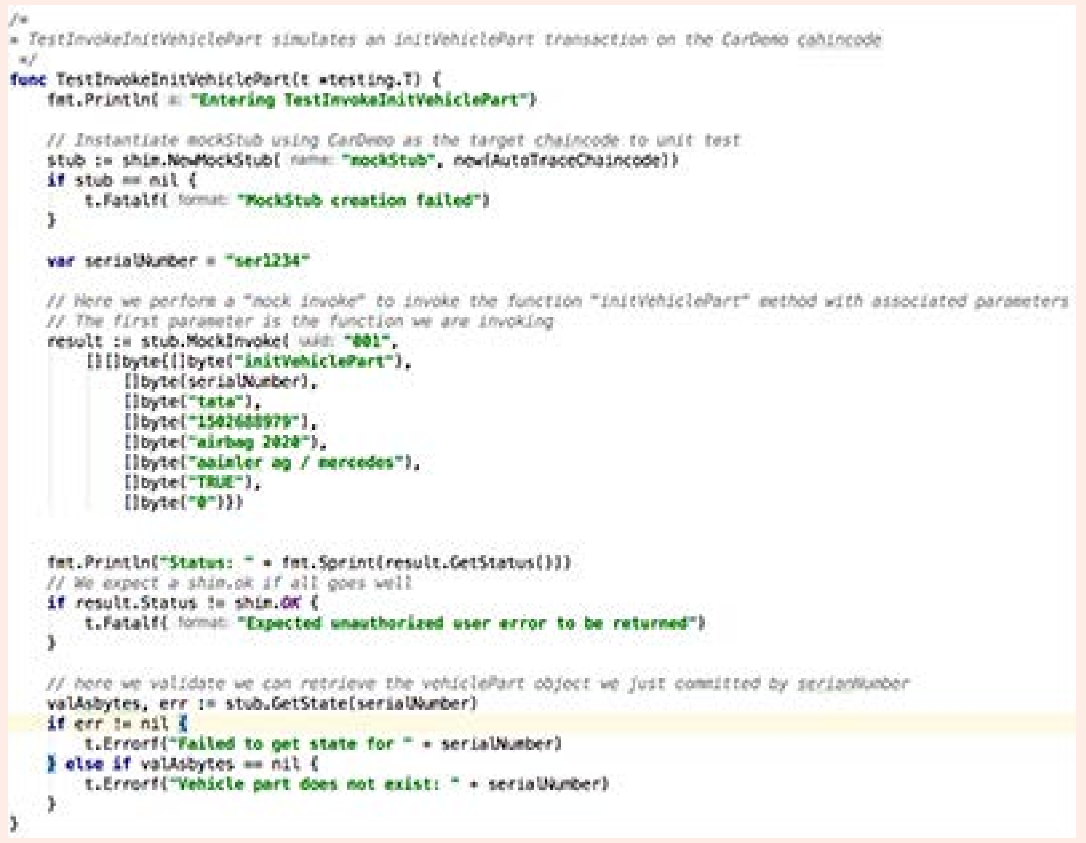


```
import (
        "testing"
        "fmt"
        "github.com/hyperledger/fabric/core/chaincode/shim"
)
```

Free SDK →

Free Cloud Trial →

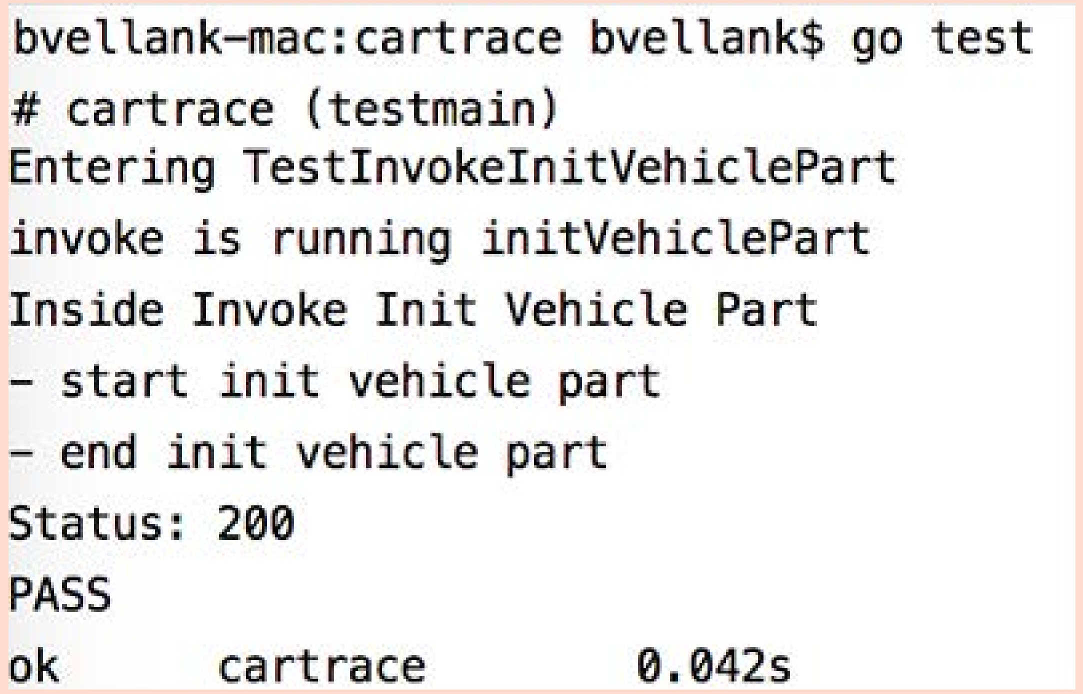
Create a unit test function for the chaincode you want to test as shown below.



```
/*
* TestInvokeInitVehiclePart simulates an initVehiclePart transaction on the CarDemo cahincode
 */
func TestInvokeInitVehiclePart(t *testing.T) {
    fmt.Println( a: "Entering TestInvokeInitVehiclePart")

    // Instantiate mockStub using CarDemo as the target chaincode to unit test
    stub := shim.NewMockStub( name: "mockStub", new(AutoTraceChaincode))
    if stub == nil {
        t.Fatalf( format: "MockStub creation failed")
    }

    var serialNumber = "ser1234"

    // Here we perform a "mock invoke" to invoke the function "initVehiclePart" method with associated parameters
    // The first parameter is the function we are invoking
    result := stub.MockInvoke( uuid: "001",
        [][]byte{[]byte("initVehiclePart"),
            []byte(serialNumber),
            []byte("tata"),
            []byte("1502688979"),
            []byte("airbag 2020"),
            []byte("aaimler ag / mercedes"),
            []byte("TRUE"),
            []byte("0")})


    fmt.Println("Status: " + fmt.Sprint(result.GetStatus()))
    // We expect a shim.ok if all goes well
    if result.Status != shim.OK {
        t.Fatalf( format: "Expected unauthorized user error to be returned")
    }

    // here we validate we can retrieve the vehiclePart object we just committed by serianNumber
    valAsbytes, err := stub.GetState(serialNumber)
    if err != nil {
        t.Errorf("Failed to get state for " + serialNumber)
    } else if valAsbytes == nil {
        t.Errorf("Vehicle part does not exist: " + serialNumber)
    }
}
```

Execute the Unit Test

1. Open the terminal window and go to the directory of the chaincode program using mock shim
2. Using your go environment to execute the test code
   a. Open a new terminal or command line window
   b. Browse to the cartrace_test.go code folder
   c. Execute from Terminal → go test (As shown in the screenshot below)



```
bvellank-mac:cartrace bvellank$ go test
# cartrace (testmain)
Entering TestInvokeInitVehiclePart
invoke is running initVehiclePart
Inside Invoke Init Vehicle Part
- start init vehicle part
- end init vehicle part
Status: 200
PASS
ok      cartrace        0.042s
```

Logging

Logging is one of the ways to detect problems in chaincode logic or data. OBP provides logging capabilities at Peer level and Chaincode level.

Peer logs can be configured through the Oracle Blockchain Cloud Service console. The logging levels can be controlled based on severity levels (CRITICAL | ERROR | WARNING | NOTICE | INFO | DEBUG).

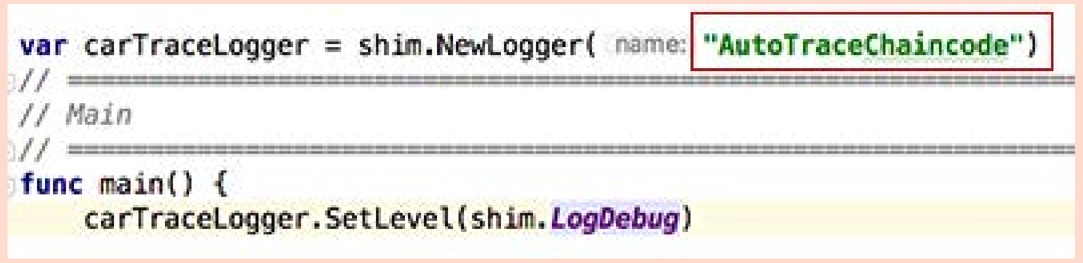
These logging levels are provided as part of peer configuration by the platform and are accessible by clicking on a peer node in the Nodes view and going to Logs view on the left hand side menu. The logs are rotated and are listed in the date order. **Current log** can be accessed by clicking on Current Log button.

Logging in chaincode is performed by integrating and exposing chaincode instance via the peer. The chaincode shim package provides API's for the chaincode. This package will allow chaincode to create and manage logging objects. These logs will be formatted and printed in the logs of the peer based on the logging levels and also accessible under Chaincodes menu once you drill down to a particular chaincode version. These logs are similarly rotated and listed in the date order. Current log can be accessed by clicking on **Current Log** button.

The following API's are available for chaincode logging:

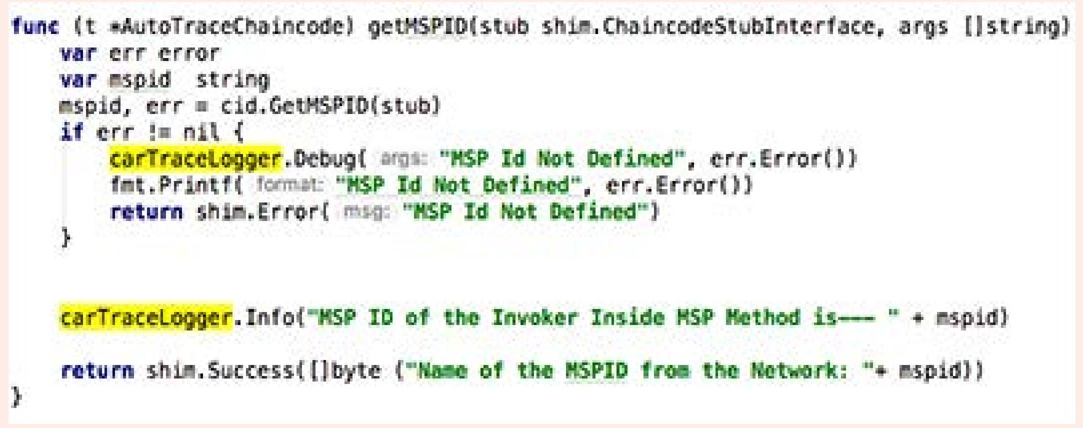
- **NewLogger(name string) *ChaincodeLogger** - Create a logging object for use by a chaincode
- **(c *ChaincodeLogger) SetLevel(level LoggingLevel)** - Set the logging level of the logger . Logging levels are LogDebug, LogInfo, LogNotice, LogWarning, LogError, LogCritical
- **(c *ChaincodeLogger) IsEnabledFor(level LoggingLevel) bool** - Return true if logs will be generated at the given level
- **LogLevel(levelString string) (LoggingLevel, error)** - Convert a string to a LoggingLevel

Example to create logging objects inside the chaincode is shown below.



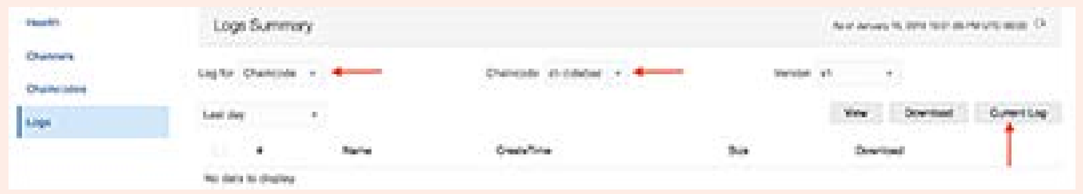
```
var carTraceLogger = shim.NewLogger( name: "AutoTraceChaincode")
// ==========================================================================
// Main
// ==========================================================================
func main() {
    carTraceLogger.SetLevel(shim.LogDebug)
```

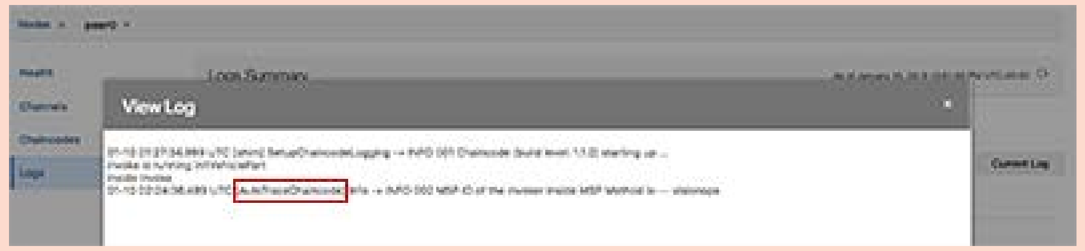
In REST format this looks like:



```
func (t *AutoTraceChaincode) getMSPID(stub shim.ChaincodeStubInterface, args []string)
    var err error
    var mspid  string
    mspid, err = cid.GetMSPID(stub)
    if err != nil {
        carTraceLogger.Debug( args: "MSP Id Not Defined", err.Error())
        fmt.Printf( format: "MSP Id Not Defined", err.Error())
        return shim.Error( msg: "MSP Id Not Defined")
    }


    carTraceLogger.Info("MSP ID of the Invoker Inside MSP Method is--- " + mspid)

    return shim.Success([]byte ("Name of the MSPID from the Network: "+ mspid))
}
```

To view the custom logger entries in OBP Peer Console, go to Nodes tab and select the peer where the chaincode is instantiated. On the peer drill-down menu select the Log option and set "Log for" selector to Chaincode, then clock **Current Log** button. In the log you will see entries prefixed with your custom logger, e.g., [AutoTraceChaincode] as shown below.





Rich Queries in Chaincode

Blockchain state DB is a key-value (KV) store. Querying KV store by value attributes rather than keys is termed "rich queries". In Hyperledger Fabric open source, there's an option to use CouchDB to support rich data queries via proprietary query language. With Oracle Blockchain Platform, the underlying KV store uses Berkeley DB (BDB), which supports SQL-based rich queries in addition to Couch DB JSON Query Language (for compatibility.) Using widely-known SQL simplifies chaincode programming, and can significantly reduce the amount of code in addition to other benefits as shown in the comparison table below.

Free SDK →

Free Cloud Trial →

Comparison of rich data queries using Oracle Berkeley DB vs Couch DB:

Berkeley DB	Couch DB
Chaincode shim API -GetQueryResults(query) → standard SQL Select statement or Couch DB Query Language format with a few limitations. Handles all functions supported by Couch DB query language plus useful SQL functions like aggregations and subqueries.	Chaincode shim API -GetQueryResults(query) → using Couchdb Query Language format with proprietary JSON grammar.
Returns an iterator with no need for pagination due to BDB's lazy evaluation model.	Returns an iterator & uses developer-specified pagination to handle large result sets.
Supports snapshot isolation levels across transactions, reducing locking and its impact on performance.	Endorsement operations can lock out commits and vice versa, impacting latency and throughput.
Query and its results included in RWset. Queries are re-run at validation time to ensure results match those at endorsement. Allows safe use of query results in state change decisions.	Results of the query are included in RWset, but the query is not re-run at the validation time. Therefore the returned KV value pairs might be different at commit time. This can lead to phantom reads problem documented in Fabric Jira FAB-2878.
Same behavior, but future plans include adding indexes without redeploying chaincode.	Indexes can be included as part of chaincode deployment.

In addition to the advantages above, Berkeley DB operates an order of magnitude faster than Couch DB due to a number of architectural differences. Berkeley DB enabled with transactions allows definition of transactional boundaries. Once committed, data is persisted to disk. To enhance performance, one can use non-durable commits, where writes are committed to in-memory log files and later synched with the underlying file systems.

With Berkeley DB there's also significantly reduced number of roundtrips to DB for complex queries due to lazy evaluations of query results.

Examples of using SQL-based rich queries in OBP chaincode:

- Query against JSON fields in values:

```
SELECT key FROM <state> WHERE json_
extract(valueJson, '$.docType')
= 'vehiclePart' AND json_
extract(valueJson, '$.owner') =
'Detroit Auto'
```

- Query using complex SQL:

```
SELECT json_extract(t.valueJson,
'$.owner') AS owner, json_extract(t.
valueJson, '$.color') AS color,
COUNT(*) AS count
FROM <STATE> t WHERE json_extract(t.
valueJson, '$.docType') = 'marble'
GROUP BY json_extract(t.valueJson,
'$.owner'), json_extract(t.
valueJson, '$.color') ORDER BY json_
extract(t.valueJson, '$.owner')
```

Returns list of owners, color, and count of each color owned, sorted by owner.

Events – Publish and Subscribe

Oracle Blockchain Platform provides publish and subscribe mechanism for event-based applications. Events are asynchronous operations that are communicated via the peer. These events can be subscribed by the entities outside the peer's organization. For example, if OBP application performs a 3-way match of PO, Invoice, and Shipping information to trigger payments, the match can be used to trigger a payment event conveyed to an ERP Financials system. OBP supports subscription to different event types via REST Proxy:

- "**transaction**": Events for a particular transaction ID
- "**txOnChannel**": Events for every new transaction on a particular channel
- "**txOnNetwork**": Events for every new transaction in the entire network
- "**blockOnChannel**": A Block header event for every new block on a particular channel

- “**blockOnNetwork**”: A Block header event on creation of a new block in the entire network
- “**chaincodeEvent**”: Custom events emitted by chaincode logic

Clients or client applications can Subscribe and Unsubscribe to all the events using the REST Proxy or Hyperledger Fabric SDKs.

Subscription REST API details are shown in the table below.

REST Endpoint	https://<rest_server_url:port/restproxy#>
Resource Path	/bcsgw/rest/v1/event/subscribe
Http Method	POST

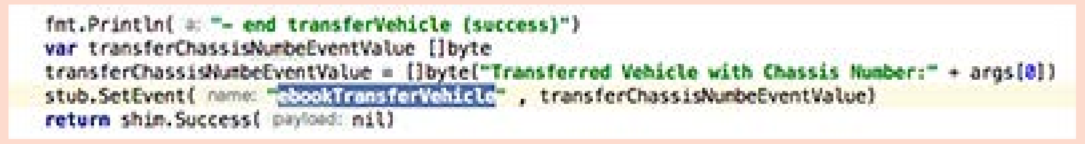
To ensure security, RESTt Proxy supports both one-way and two-way SSL authentication to connect Proxy with client callback server. To access one-way SSL authentication, client needs to pass the callback server’s CA certificate, and a valid client certificate (only for mutual authentication) to the REST Proxy.

For two-way SSL authentication (mutual TLS), callback server’s CA should issue a new certificate and private key for Rest Proxy. This would be included as part of the callback parameters in the Subscription REST API call.

Publish Events:

Unlike Chaincode events all the other event types like transaction & block events are provided by the framework and do not require any programming to publish the events. Chaincode events are published by setting up the events (programming) in chaincode using **stub.SetEvent()** method and deploying the chaincode on to the network.

Example:

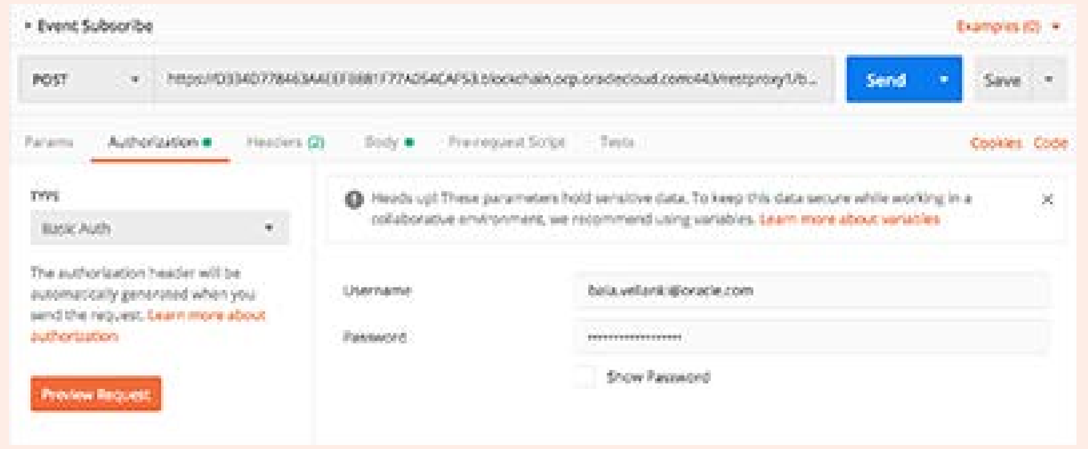


```
fmt.Println( a: "- end transferVehicle (success)")
var transferChassisNumbeEventValue []byte
transferChassisNumbeEventValue = []byte("Transferred Vehicle with Chassis Number:" + args[0])
stub.SetEvent( name: "ebookTransferVehicle" , transferChassisNumbeEventValue)
return shim.Success( payload: nil)
```

Steps to Subscribe to Events:

Most useful type of events for subscribing are chaincode events. There is also a good use of subscribing to block and transaction events, but majority of the useful events are chaincode events. Let’s walkthrough step by step to subscribe to events using Rest Proxy API’s. Developers can use Postman or other tools to perform RESTful API calls. Screenshots below for performing REST API use Postman.

1. Setup authentication using Basic Auth.



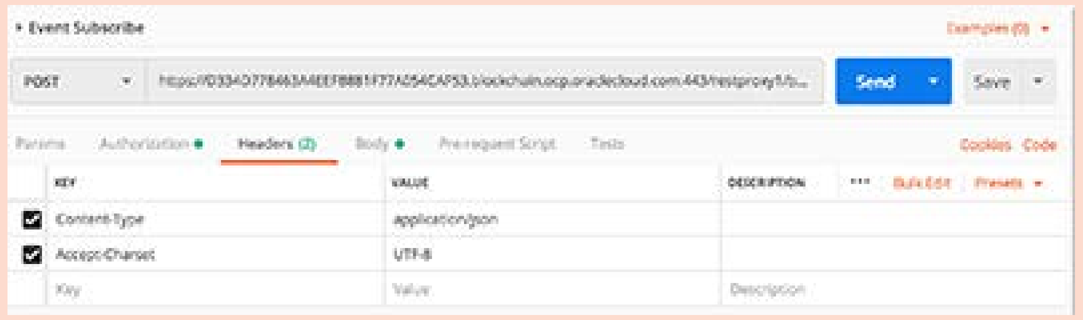
Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables

The authorization header will be automatically generated when you send the request. Learn more about authorization

Username: bala.vellanki@oracle.com

Password: ••••••••••••••••

2. Setup http headers for subscribing events.



POST [illegible]

	KEY	VALUE	DESCRIPTION
✓	Content-Type	application/json	
✓	Accept-Charset	UTF-8	
	Key	Value	Description

3. Setup the request body to subscribe for the events by configuring the callback URL and other required parameters in the JSON body of the http POST request as shown below.

```
{
  “requests”: [
    {
      “eventType”: “chaincodeEvent”,
      “callbackURL”: “http://ebookclientapp-webhook/
evtSender1”,
      “callbackTlsCerts”: {
          “caCert”: “-----BEGIN CERTIFICATE-----
\....\n-----END CERTIFICATE-----”,
          “clientCert”: “-----BEGIN CERTIFICATE-
----\....\n-----END CERTIFICATE-----\n-----BEGIN
ENCRYPTED PRIVATE KEY-----\....\n-----END ENCRYPTED
PRIVATE KEY-----\n”,
          “keyPassword”: “T3JhY2xl”
        },
      “expires”: “1m”,
      “chaincode”: “cartrace”,
      “eventName”: “ebookTransferVehicle”,
      “channel”: “ebookchannel”
    }
  ]
}
```

The **eventName** corresponds to the name used in **setEvent()** API in the chaincode. The expires field specifies the time until event subscription expires and can be provided in one of the formats below:

xxM:	months
xxw:	weeks
xxd:	days
xxh:	hours
xxm:	minutes

The callbackTlsCerts structure contains one mandatory file and two optional fields:

- caCert (mandatory): and is the callback server’s CA certificate in PEM format. It will be verified by the REST proxy before registering the subscription.
- clientCert (optional): refers to the certificate REST proxy should use during the callback. It’s only needed when mutual authentication is required. It must be in PEM format and assumed the certificate and private key are concatenated. If you have PKCS#12 format certificate, use Linux command.

  `openssl pkcs12 -in client.p12 -out client.pem` to convert it to PEM format.
- keyPassword (optional): only required when client cert is concatenated with an encrypted private key. If used it should be base64 encoded.

4. If the subscription was successful, developer will receive a subscription id.

```
“response”: {
        “returnCode”: “Success”,
        “subid”: “75d8cc12-7673-4712-
ba42-715adb69e6b1”
      }
```

5. After subscription, the callback URL mentioned in the body of the subscription will receive all the events generated when the chaincode is executed with even name as specified and event payload message used in setEvent() API in the chaincode:

```
{
  “eventType”: “chaincodeEvent”,
  “subid”: “<UUID for the event
subscription>”,
“channel”: “ebookChannel”,
  “eventMsg”: {
    “chaincodeId”: “cartrace”,
    “txId”: “asdfgshjyjnmytrbndxvdfg
txid”,
    “eventName”: “ebookTransferVehicle”,
    “payload”: {
      “type”: “UTF-8”,
      “data”: [“porsche1000001”, “john-
dealership”, “clark-and-sons”]
    }
  }
}
```

If you are calling back into a custom application, it will need to parse this response and extract relevant payload information.

If your callback is for a packaged application that has its own REST API format, you can use REST connector in Oracle Integration Cloud (OIC) to map the callback data to the desired inbound REST call.

Steps to Un-Subscribe to Events:

Endpoint URL	https://<rest_server_url:port/restproxy#>
Resource Path	/bcsgw/rest/v1/event/unsubscribe
Http Method	POST

1. Follow steps 1 & 2 from subscribe events to retrieve the subscription id

2. Setup http headers for subscribing events similar to subscribe

3. Setup the **request** body as shown below to Unsubscribe an event using its subscription ID

Request body to Unsubscribe

```
{
  “request”:
    {
      “subid”: “75d8cc12-7673-4712-ba42-
715adb69e6b1”
    }
}
```

Response for Unsubscribe Call

```
{
“response”:
 {
“returnCode”: “Success”,
“subid”: “f5505ce8-54c5-41c7-8ae8-
8077f630af9b”
         }
```

Using Rich History Database for Analytics/BI

The transaction history in Hypeledger Fabric and OBP is maintained in the linked blocks of transactions stored in the peer's filesystem and indexed by history DB pointers. When accessing transaction history, chaincode uses a shim API to request transactions for particular key, and peer nodes retrieve this information based on the history DB pointers into the blocks. While this works well for retrieving transaction sequence for a few keys, it is a cumbersome way to aggregate data for analytics. You are also forced to retrieve all the transactions in the history sequentially rather than being able to only retrieve the ones matching certain attributes, like you can do with rich queries for world state database. Creating reports on analytics dashboards would require many query transactions creating additional load on the network. Another alternative is to have a client subscribe to all new block events and have the client applications parse the blocks and load their data into a data warehouse.

In REST format this looks like:

In Oracle Blockchain Platform, neither of these are necessary. OBP provides a built-in mechanism to shadow rich history updates to an Oracle database service, such as Autonomous Data Warehouse or DBaaS that can be specified by the user. This can be configured per channel, and the database would contain state and transaction history for the selected channel's ledger. This configuration option can be specified as a checkbox in Create a New Channel wizard or selected from the channel's actions menu on Channels tab. When enabled, the peer will asynchronously write the latest state and history data to the associated Oracle DB as transactions are committed by the peers. Please refer to the Oracle schema documentation for how state and history data is mapped in RDBMS. Note the use of JSONValue fields, which can be unpacked into their own relational columns using JSON support in Oracle Database. You can use any BI tools to create reports and interactive dashboards for visualization about the data in your ledger based on rich history DB.

For example, using the rich history database, you could create an analytics report to learn the average account balance of all of the customer accounts over some time interval, or an interactive dashboard to monitor how long it takes to ship different types of merchandise from a wholesaler to a retailer.

Developers can use an Oracle database such as Oracle Autonomous Data Warehouse (ADW) or any other Oracle DBaaS to create your rich history database. Once created, any BI tools or cloud services, such as Oracle Analytics Cloud (OAC) can be used to create reports and dashboards.

In this configuration setup, we will be using DBaaS with OAC service. You will need DBaaS connection string parameters or a wallet to setup the history DB connections in OBP and OAC.

Steps to configure and use rich history DB:

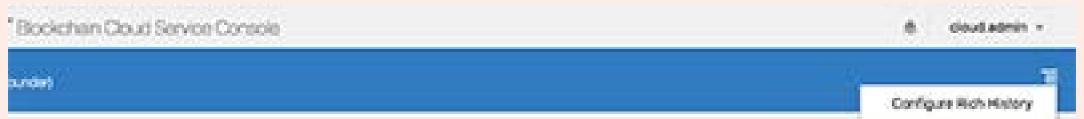
1. Create a DB service in Oracle Cloud, such as Autonomous Data Warehouse or DBaaS.

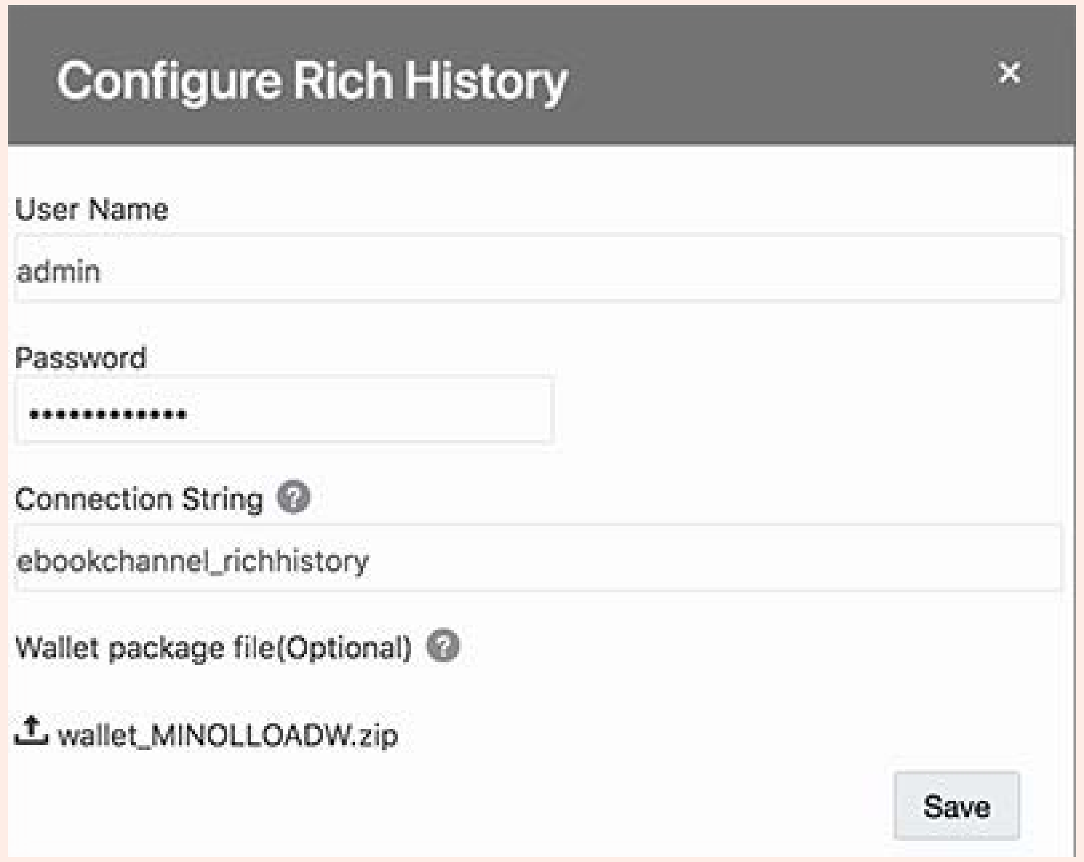
In REST format this looks like:

Free SDK →

Free Cloud Trial →

2. In OBP use the Actions menu on the right of the blue service bar and select Configure Rich History. Fill in DB connection information as shown below.





Configure Rich History

User Name
admin

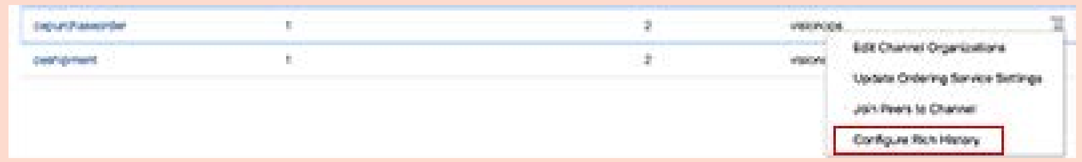
Password
••••••••••••

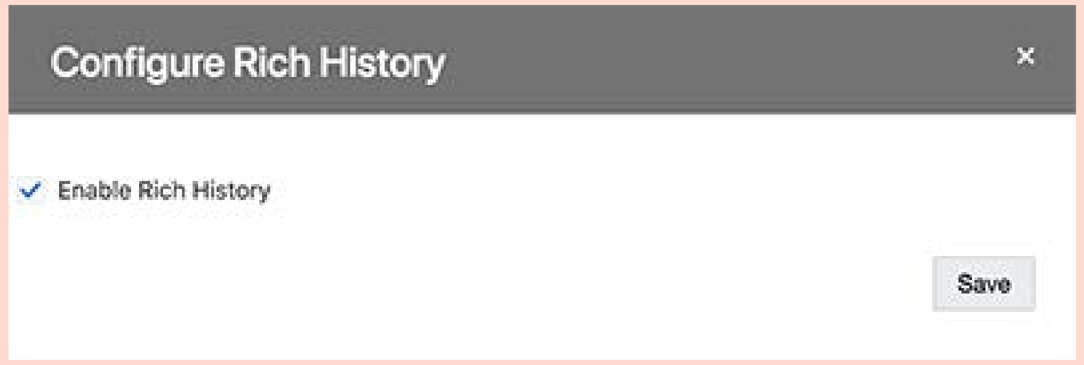
Connection String
ebookchannel_richhistory

Wallet package file(Optional)
wallet_MINOLLOADW.zip

Save

3. Once the connection is established, switch to the Channels tab and use a channel's action menu to select Enable Rich History for that channel.





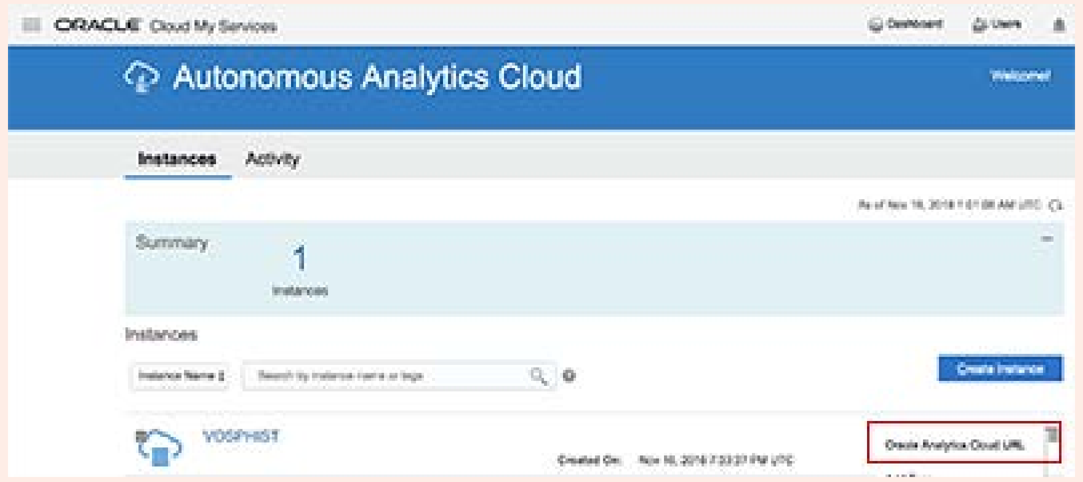
Configure Rich History

✓ Enable Rich History

Save

That's all you need to do. Now when any transactions are committed on this channel, OBP will automatically shadow them into a pre-defined relational schema in the associated Oracle database. Run a few transactions and check that your database is getting an update.

4. Now let's do some analytics. You can use any BI tools, but we will create Oracle Analytics Cloud (OAC) instance, which you can do from MyServices dashboard. Once an OAC instance is ready, use the actions menu to select Oracle Analytics Cloud URL option to open up the OAC home page.

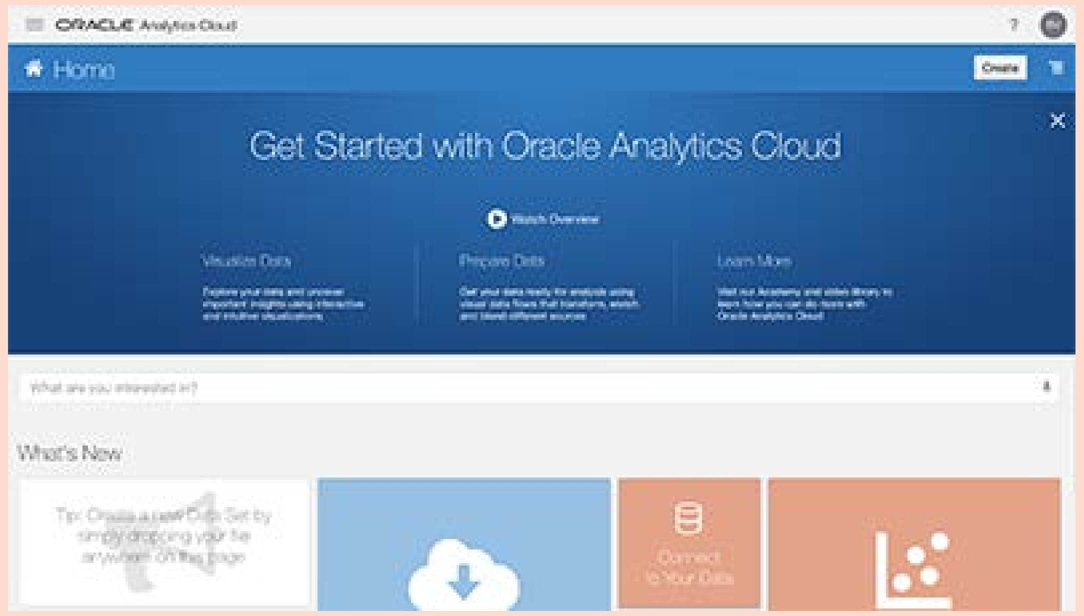


5. Complete documentation for OAC is available at https://docs.oracle.com/en/cloud/paas/analytics-cloud/index.html, where you can refer to detailed instructions on defining connections, datasets, and creating visualizations. The material below provides a basic illustration of these steps using the cartrace rich history database as an example.

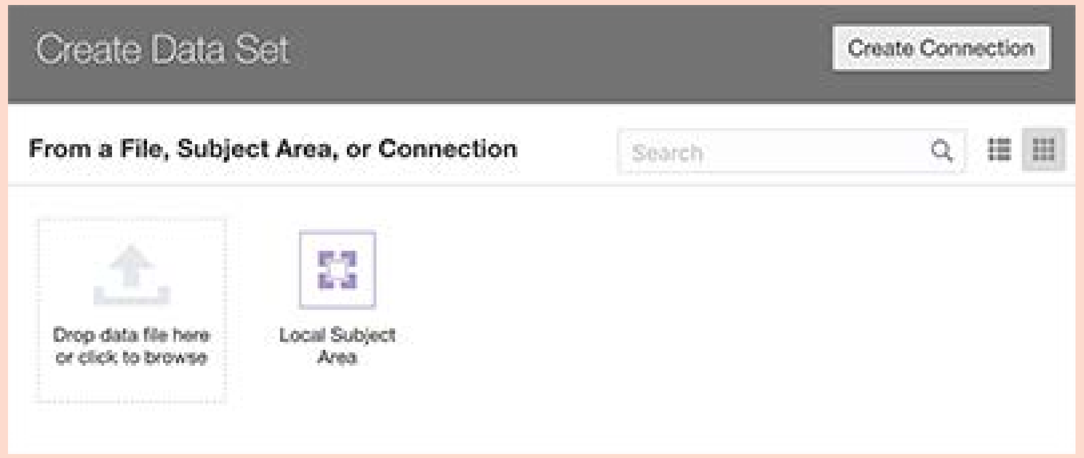
In REST format this looks like:

Free SDK →

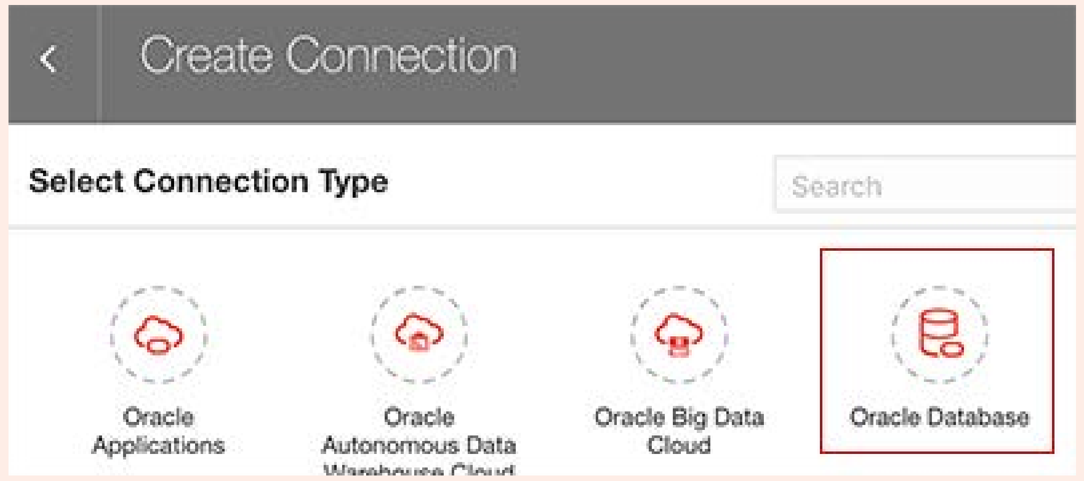
Free Cloud Trial →

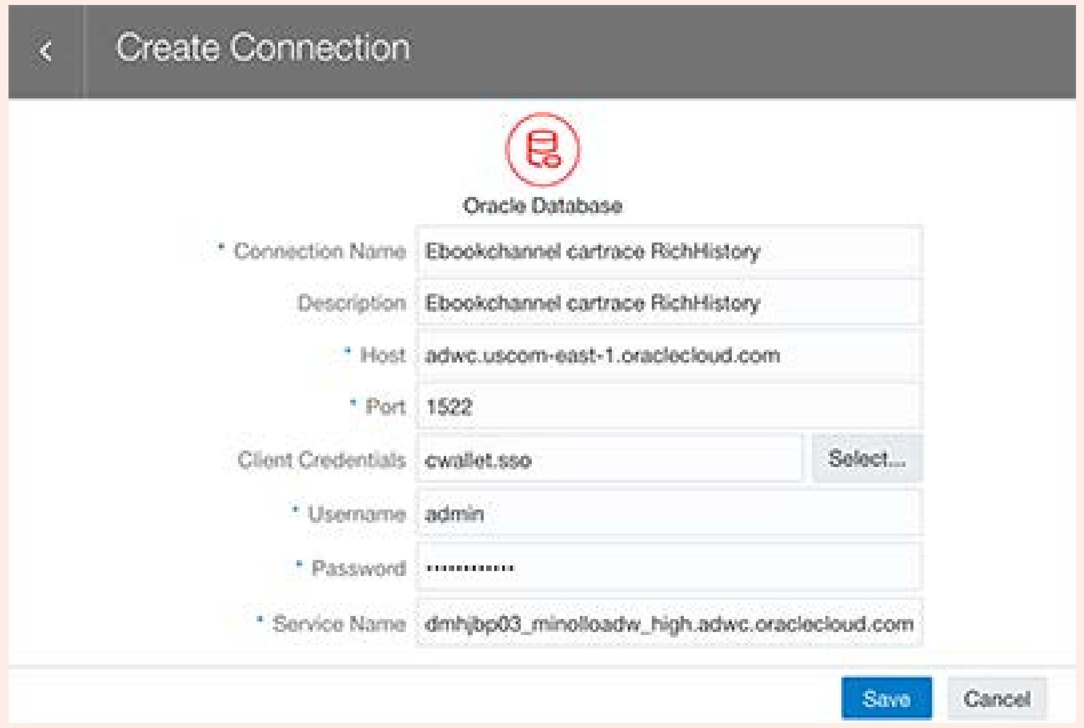


6. Click **Create Connection** button to configure the Analytics service to connect with DBaaS Service created in step 1.


Create Data Set
Create Connection
From a File, Subject Area, or Connection
Search
Drop data file here or click to browse
Local Subject Area

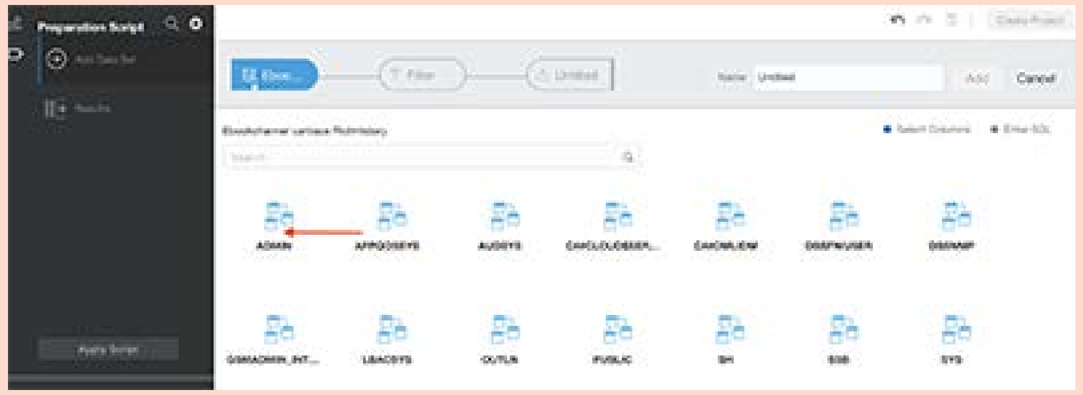
Select Oracle Autonomous Data Warehouse Cloud or Oracle Database depending on which kind of DBaaS service you are using and fill in connection information.





* Connection Name	Ebookchannel cartrace RichHistory
Description	Ebookchannel cartrace RichHistory
* Host	adwc.uscom-east-1.oraclecloud.com
* Port	1522
Client Credentials	cwallet.sso
* Username	admin
* Password	•••••••••••••
* Service Name	dmhjbp03_minolloadw_high.adwc.oraclecloud.com

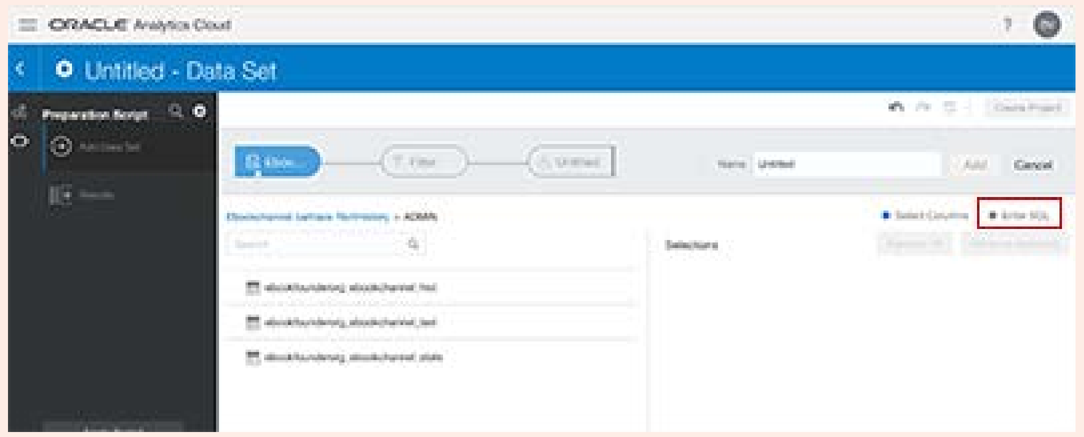
7. Select the ADMIN user profile after the connection is successful.



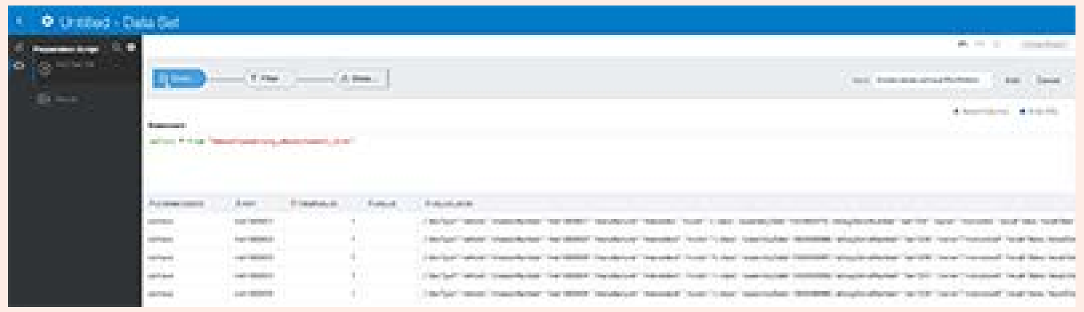
You will see the history and state tables:

- **<OBP instance>_<OBP channel>_hist** for transaction history data
- **<OBP instance>_<OBP channel>_state** for world state data

There's also an <OBP instance>_<OBP channel>_last table shown. You can't query this table for analytics since it's used internally by OBP for tracking the block height recorded in the rich history database. It determines how current the rich history database is and if all of the chaincode transactions were recorded in the rich history database.



8. Click Enter SQL selector on the right to perform a SQL query to view the contents of history or state tables. Start with **select * from "<OBP instance>_<OBP channel>_hist"** substituting **ebookfounderorg** for OBP instance and **ebookchannel** for OBP channel.



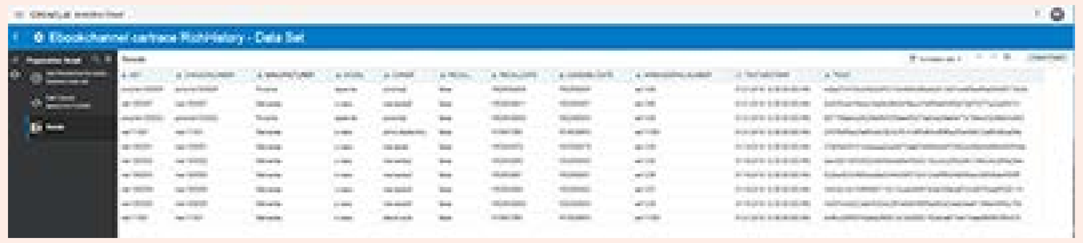
Note the VALUEJSON column that contains all the interesting attributes of the transaction. To enable full use of this data in your analytics dashboard or report, this column will need to be unpacked into relational format when defining your dataset. Oracle database provides built-in JSON support that makes this trivial.

9. Replace the select above with the one below to extract JSON values into relational form along with a few other fields:

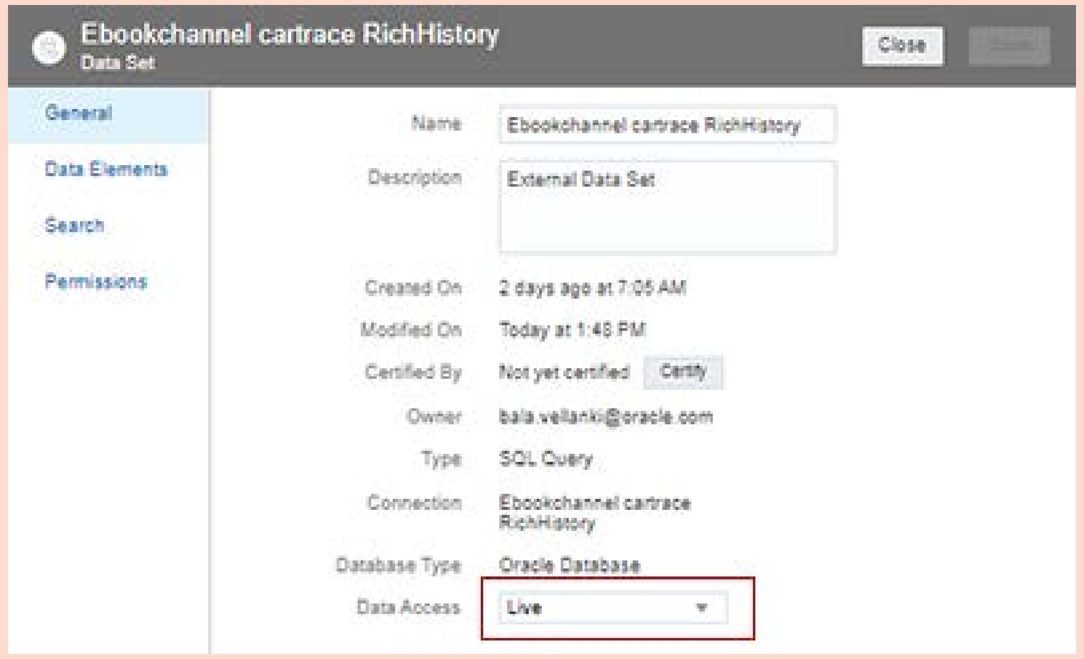
```
select h.key,
h.valueJson.chassisNumber,
h.valueJson.manufacturer,
h.valueJson.model,
h.valueJson.owner,
h.valueJson.recall,
h.valueJson.recallDate,h.valueJson.
assemblyDate,
h.valueJson.airbagSerialNumber,
h.TXNTIMESTAMP,
h.TXNID
from "ebookfounderorg_ebookchannel_hist" h
where h.valueJson.manufacturer IS NOT NULL
```

Note the use of "h" as an alias in the from clause to enable us to refer to JSON fields in the form of **<table-alias.column-name.field>**.

The result dataset might look like the one shown below and is now ready to be used in interactive dashboards or visualizations as they are called in OAC.



10. Define a project and create visualizations based on this dataset. In a project view you can right click the dataset and select Inspect option to see various details. Note the last field enables you to switch between Live access to data (as its being updated from the blockchain peer node) or Cached mode.

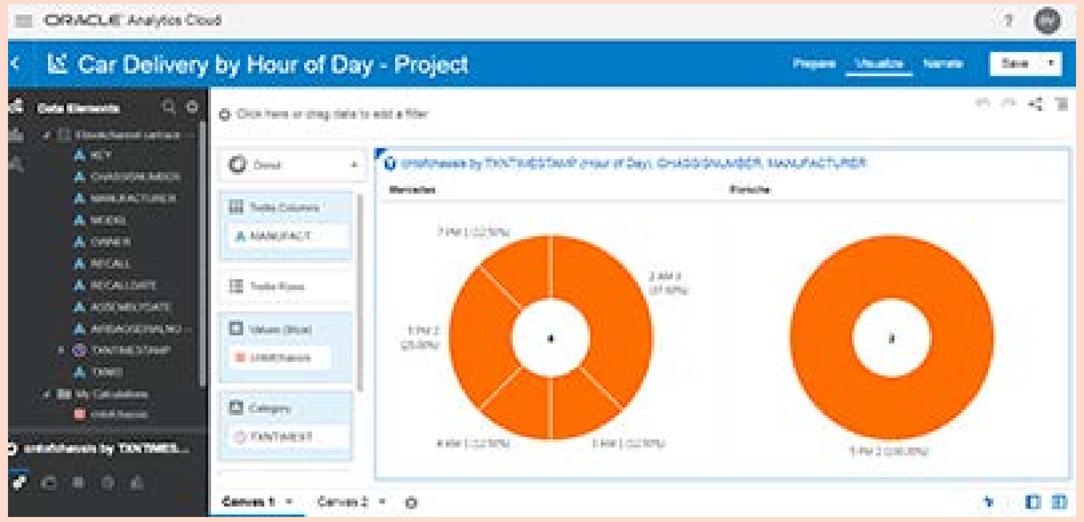


Ebookchannel cartrace RichHistory
Data Set
Close

General
Data Elements
Search
Permissions

Name	Ebookchannel cartrace RichHistory
Description	External Data Set
Created On	2 days ago at 7:05 AM
Modified On	Today at 1:48 PM
Certified By	Not yet certified Certify
Owner	bala.vellanki@oracle.com
Type	SQL Query
Connection	Ebookchannel cartrace RichHistory
Database Type	Oracle Database
Data Access	Live

As an example, let's create some dashboards counting the number of vehicles added to the inventory by hour of day based on transaction timestamp and grouped by the manufacturer using a Donut visualization style.



To make this visualization using a few clicks we:

1) Defined an aggregate cntofchassis under **My Calculations**

2) Selected Donut visualization for the Canvas

3) Dragged the following fields into the Donut components:

   a. **TXNTIMESTAMP** to Category
   b. **cntofchassis** to Values
   c. **MANUFACTURER** to Trellis Columns

4) Customized the chart using controls on the bottom left

The same dataset can be used for multiple visualizations. Let's create another one where we are looking for how many vehicles have been delivered to different dealerships (owners) by manufacturers.

To make this visualization using a few clicks we:

1) Added a new Canvas at the bottom

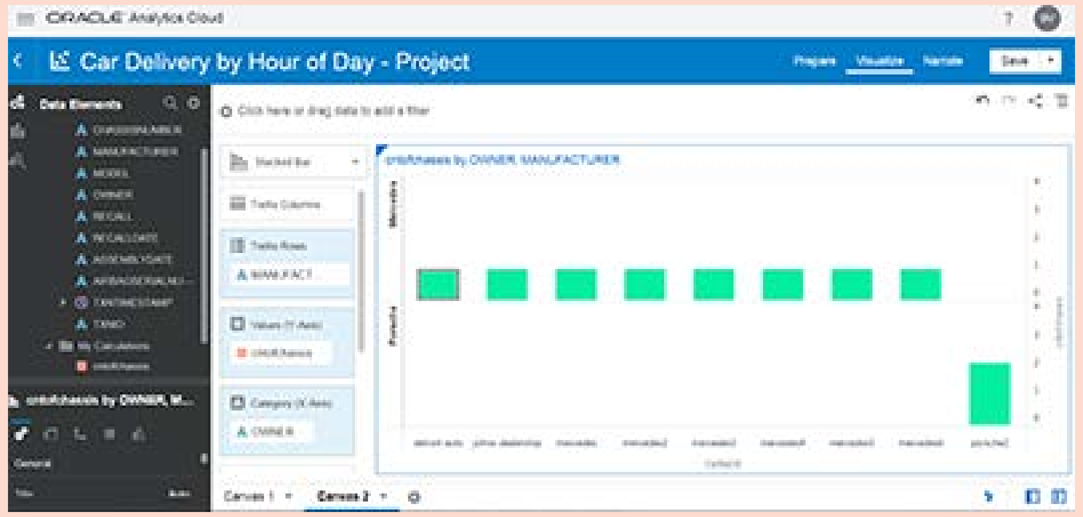
2) Selected Stacked Bar visualization for this Canvas

3) Dragged the following fields into the Stacked Bar components:

   a. **OWNER** to Category (X-Axis)
   b. **cntofchassis** to Values (Y-Axis)
   c. **MANUFACTURER** to Trellis Rows

4) And again, customized the chart using controls on the bottom left

The results are shown in the screenshot below.



Many powerful analytics capabilities can be supported using rich history database enabled by OBP, and in particular if this data is used in conjunction with other data sources.

Free SDK →

Free Cloud Trial →

Summary

The Oracle Blockchain Platform is a comprehensive blockchain platform for building trusted networks that securely and reliably accelerate business-to-business transactions. This production-ready service includes admin console, membership service, peer nodes, orderer service, and REST proxy. Once provisioned, you can link instances into a network and begin to build custom chaincodes to extend business applications such as payments, contracts, invoicing, shipping, and accounting. Integrate blockchain with existing applications via REST APIs or client SDKs, and publish events, trigger notifications, or incorporate existing business logic. Easily control access and privileges with channels that facilitate scalability and enable confidentiality. With Oracle Blockchain Platform you can quickly and securely connect your organization to others in your ecosystem for a wide range of automated business transactions.

References and Resources

- Oracle Blockchain Platform product page
- Oracle Blockchain Platform developer site
- Oracle Blockchain Platform documentation
- Hyperledger Fabric open source documentation
- Oracle Blockchain News & Opinion
- Oracle Blockchain Platform Getting Started video
- Oracle Blockchain Platform Tutorials
- Downloadable free Oracle Blockchain Platform SDK for time-limited development and test

developer.oracle.com

ORACLE®