Oracle® Cloud

Using Oracle Blockchain Platform

F26726-43
May 2026

Oracle Cloud Using Oracle Blockchain Platform,

F26726-43

Contents

4 Understand and Manage Nodes by Type

5 Extend the Network

6 Develop Chaincodes

7 Deploy and Manage Chaincodes

8 Develop Blockchain Applications

9 Work With Databases

About This Content

Oracle Blockchain Platform includes all the dependencies required to support a permissioned blockchain network.

Audience

This guide is intended for anyone who uses Oracle Blockchain Platform.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Conventions

The following text conventions are used in this document.

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

What's Oracle Blockchain Platform?

This topic contains information to help you understand what Oracle Blockchain Platform is.

Topics:

What's a Blockchain?

A blockchain is a system for maintaining distributed ledgers of facts and the history of the ledger’s updates. A blockchain is a continuously growing list of records, called blocks, that are linked and secured using cryptography.

This allows organizations that don't fully trust each other to agree on the updates submitted to a shared ledger by using peer to peer protocols rather than a central third party or manual offline reconciliation process. Blockchain enables real-time transactions and securely shares tamper-proof data across a trusted business network.

A blockchain network has a founder that creates and maintains the network, and participants that join the network. All organizations included in the network are called members.

Oracle Blockchain Platform is a permissioned blockchain, which provides a closed ecosystem where only invited organizations (or participants) can join the network and keep a copy of the ledger. Permissioned blockchains use an access control layer to enforce which organizations have access to the network. The founding organization, or blockchain network owner, determines the participants that can join the network. All nodes in the network are known and use consensus protocol to ensure that the next block is the only version of truth. There are three steps to consensus protocol:

- **Endorsement —** This step determines whether to accept or reject a transaction.
- **Ordering —** This step sorts all transactions within a time period into a sequence or block.
- **Validation —** This step verifies that the required endorsement are gotten in compliance with the endorsement policy and organization permissions.

Blockchain's key properties

Shared, transparent, and decentralized— The network maintains a distributed ledger of facts and update history. All network participants see consistent data. Data is distributed and replicated across the network’s organizations. Any authorized organizations can access data.

Immutable and irreversible — Each new block contains a reference to the previous block, which creates a chain of data. Data is distributed among the network organizations. Blockchain records can only be appended and can't be undetectably altered or deleted. Consensus is required before blocks or transactions are written to the ledger. Therefore, the existence and validity of a data record can't be denied. After endorsement policies are satisfied and consensus is reached, data is grouped into blocks and blocks are appended to the ledger with

cryptographically secured hashes that provide immutability. Only those members authorized to have the corresponding encryption keys can view data.

Encryption — All records are encrypted.

Closed ecosystem — Joined organizations can have a copy of the ledger. Organizations are known in the real world. Consensus protocols depend on knowing who the organizations are.

Speed — Transactions are verified in minutes. Network members interact directly.

Blockchain example

An example of an organization that benefits from using blockchain is a supply chain contract manufacturing company. Suppose this company is located in the United States and uses a third-party company in Mexico to source materials for and produce electronic components. With a blockchain network, the manufacturing company can quickly know the answers to the following questions:

- Where is the product in the production cycle?
- Where is the product being produced?
- Does the product contain ethically sourced materials?
- Does the product meet specifications and exporting compliance rules?
- When is ownership transferred?
- Does the invoice match and should the organization pay it?
- How should the organization handle any exceptions to the manufacturing, shipping, or receiving process?

Why Should I Use Blockchain?

Implementing blockchain can help you manage and bring efficiency to many aspects of your business practices.

The key benefits of using a blockchain are:

Increase Business Velocity — You can create a trusted network for business-to-business transactions and extend and automate your operations beyond the enterprise. With blockchain, you can optimize business decisions by providing real-time information visibility across your company's ecosystem.

Reduce Operation Costs — Use blockchain to accelerate transactions and eliminate cumbersome offline reconciliations by using a trusted shared fabric of common information. Blockchains help you eliminate intermediaries and related costs, possible single points of failure, and time delay by using a peer to peer business network.

Reduce the cost of fraud and regulatory compliance — Blockchain enables you to gain the security of knowing that business critical records are made tamper-proof with securely replicated, cryptographically linked blocks that protect against single point of failure and insider tampering.

What Are the Advantages of Oracle Blockchain Platform?

Using Oracle Blockchain Platform to create and manage your blockchain network has many advantages over other available blockchain products.

As a preassembled PaaS, Oracle Blockchain Platform includes all the dependencies required to support a blockchain network: compute, storage, containers, identity services, event services, and management services. Oracle Blockchain Platform includes the blockchain network console to support integrated operations. This helps you start developing applications within minutes, and enables you to complete a proof of concept in days or weeks rather than months.

How Oracle Blockchain Platform Adds Value to Hyperledger Fabric

Oracle Blockchain Platform is based on the Hyperledger Fabric project from the Linux Foundation, and it extends the open source version of Hyperledger Fabric in many ways.

Provisioning and Integration in Oracle Cloud Infrastructure

- Includes preassembled PaaS with template-based provisioning. See Before You Create Your Instance.
- Uses Oracle Cloud Infrastructure to incorporate infrastructure dependencies (managed containers, virtual machines, identity management, block and object storage).
- Supports multi-cloud, hybrid blockchain network topology that spans multiple Oracle Cloud Infrastructure data centers, on-premises deployments of Hyperledger Fabric, and third-party clouds to link blockchain nodes across organizations, data centers, and continents.

Operates as an Oracle Managed Service

- Includes Oracle operations monitoring.
- Has zero-downtime managed patching and updates.
- Includes embedded ledger and configuration backups.

Enhances Security

- Uses data in-transit encryption based on TLS 1.3 or TLS 1.2, prioritizing forward-secrecy ciphers in the TLS cipher-suite.
- Uses data at-rest encryption for all configuration and ledger data.
- Isolates customers from other tenants and from Oracle staff.
- Includes a web application firewall to protect blockchain components against cyberattacks, including predefined Open Web Access Security Project (OWASP) rules, aggregated threat intelligence from multiple sources, and layer 7 distributed denial-of-service (DDoS) attacks.
- Provides audit logging of all API calls to the blockchain resources, with records available through an authenticated, filterable query API or as batched files from Oracle Cloud Infrastructure Object Storage.

Leverages Built-In Oracle Identity Cloud Service Integration

- Provides user and role management. See Set Up Users and Application Roles.
- Provides authentication for the Oracle Blockchain Platform console, REST Proxy, and certificate authority (CA).

- Supports identity federation and third-party client certificates to enable consortia formation and simplify member onboarding.

Adds REST Proxy

- Supports a rich set of Fabric APIs through REST calls for simpler transaction integration. See REST API for Oracle Blockchain Platform.
- Enables synchronous and asynchronous invocations. Enables events and callbacks and DevOps operations.
- Simplifies integration and insulates applications from underlying changes in transaction flow.

Accelerates Integration

- Provides plug-and-play enterprise adapters using Oracle Integration Cloud Service to integrate Oracle SaaS, PaaS, and on-premises applications with blockchain transactions, queries, and events. See Oracle Integration.
- Blockchain-enabled Oracle Flexcube, Open Banking API Platform, and other Oracle applications with embedded blockchain APIs.
- Enables ERP, EPM, GL, SCM, and HCM business processes in Oracle SaaS, on-premises, and non-Oracle systems to rapidly integrate with blockchain to streamline data exchange and conduct trusted transactions with other organizations.

Provides the Management and Operations Console

- Provides a comprehensive, intuitive web user interface and wizards to automate many administration tasks. For example, adding organizations to the network, adding new nodes, creating channels, deploying chaincodes, browsing the ledger, and more. See Oracle Blockchain documentation library.
- Enables DevOps through REST APIs for administration and monitoring of the blockchain.
- Dynamically handles configuration updates without a node restart.
- Includes dashboards, a ledger browser, and log viewers for monitoring and troubleshooting.

Replaces Ledger DB World State Store With Oracle Berkeley DB

- Provides Couch DB rich query support at Level DB performance.
- Provides SQL-based rich query support. See What's the State Database?
- Validates query results at commit time to ensure ledger integrity and avoid phantom reads.

Integrates Rich History Database

- Enables transparent shadowing of transaction history and private data collections to Autonomous AI Lakehouse or Database as a Service and the use of Analytics or Business Intelligence (for example, Oracle Analytics Cloud or third-party tools) on blockchain transaction history and world state data. See Create the Rich History Database.
- Supports standard tables and blockchain tables for storing rich history. Blockchain tables are tamperproof append-only tables, which can be used as a secure ledger while also being available for transactions and queries with other tables.

Includes Low-Code Blockchain App Builder

Blockchain App Builder assists with rapid development, testing, debugging, and deployment of chaincode on Oracle Blockchain Platform networks. Blockchain App Builder generates complex chaincodes in TypeScript (for Node.js chaincode) and Go (for Go chaincode) from a simple specification file. Blockchain App Builder supports the full development life cycle either from a command-line interface or as an extension for Visual Studio Code.

Blockchain App Builder also includes tokenization support for both fungible and non-fungible tokens. Token classes and methods are automatically generated, and additional token methods are provided so that developers can create complex business logic for tokens.

For more information, see Blockchain App Builder for Oracle Blockchain Platform.

Supports Hybrid State Database Model

Peer nodes can configure Oracle Database as a fallback state database. The hybrid state database model avoids service interruptions by storing the state information in both the embedded Berkeley DB (primary) and also in Oracle Database (fallback). If an issue arises with the primary state database, Oracle Blockchain Platform automatically switches to the fallback state database while the primary state database recovers.

Highly Available Architecture and Resilient Infrastructure

Built for business-critical enterprise applications, Oracle Blockchain Platform is designed for continuous operation as a highly secure, resilient, scalable platform. This platform provides continuous monitoring and autonomous recovery of all network components based on continuous backup of the ledger blocks and configuration information.

Each customer instance uses a framework of multiple managed VMs and containers to ensure high availability. This framework includes:

- Peer node containers distributed across multiple VMs to ensure resiliency if one of the VMs is unavailable or is being patched.
- Orderers, Fabric CA, console, and REST proxy nodes are replicated in all VMs for transparent takeover to avoid outages.
- Isolated VM environments for customer chaincode execution containers for greater security and stability.

Built-in integration with Oracle Identity Cloud Service for user authentication, roles management, and identity federation immediately leverages Oracle Identity Cloud Service accounts and enables onboarding of consortium members who prefer using SAML-based federation for authentication against their own identity providers.

Oracle Blockchain Platform is an Oracle managed service in which provisioning, running, and maintaining all of the infrastructure is transparent to customers. The entire framework can be provisioned with only a few clicks and user inputs, such as which shape to use, the initial number of peers, and if the instance type is Founder or Participant. The rest of the instance is automatically defined by the QuickStart shape that you select. See Before You Create an Oracle Blockchain Platform Instance.

The platform is integrated with Oracle Cloud operations management and monitoring service for continuous DevOps. Full stack zero-downtime patching and upgrades are provided with the platform. These are transparently performed by Oracle operations with no customer downtime required. If any security vulnerabilities are discovered, emergency security patching is enabled for the operating system and all of the components that the service comprises. Ongoing adaptive intelligent cyber-threat detection, mitigation, and remediation are provided as part of the Oracle Cloud Infrastructure security-in-depth approach. This leverages machine learning-

based adaptive intelligence for quick detection of intrusions and abnormal behaviors, and automated patching as one of the tools for faster remediation. See Oracle Cloud Infrastructure Documentation.

Oracle Blockchain Platform supported by Oracle Cloud Infrastructure and Oracle Cloud Operations delivers the best-in-class levels of availability, performance, and security. For availability SLAs, see Oracle PaaS and IaaS Public Cloud Services - Pillar Document.

What Do I Get with Oracle Blockchain Platform?

Your instance includes everything you need to build, run, and monitor a complete production-ready blockchain network based on Hyperledger Fabric.

Your Oracle Blockchain Platform instance is defined by the shape and the underlying platform version of Hyperledger Fabric that you selected when you created your instance. See Before You Create Your Instance. Your instance includes validating peer nodes, a membership services provider (MSP), and an ordering service.

For more information about chaincode lifecycle management, see Typical Workflow to Deploy Chaincodes.

In addition, REST proxy nodes are provided and a default channel is created. Use the console user interface to further configure, administer, and monitor the network, as well as install, deploy, and upgrade smart contracts (also known as chaincodes). The Developer Tools tab contains sample chaincodes that you can deploy and run to help you quickly understand how the blockchain network works.

Because Oracle Blockchain Platform is part of the Oracle Cloud platform, it's pre-assembled with the underlying cloud services, including containers, compute, storage, identity cloud services for authentication, object store for embedded archiving, and management and log analytics for operations and troubleshooting. You can configure multiple peer nodes and channels for availability, scalability, and confidentiality, and Oracle Cloud will automatically handle the underlying dependencies.

Additional instances can be created for other organizations and joined into your blockchain network. As an Oracle Cloud service, each instance includes replicated resources, monitoring and recovery agents, embedded archiving of configuration data and ledger blocks, and integration with management and log analytics services to help Oracle operations monitor and troubleshoot any issues. It also includes zero-downtime managed patching to resolve any issues and upgrade the service components without interrupting your operations.

2

Get Started Using Samples

This topic contains information about the samples included in your instance. Using samples is the fastest way for you to get familiar with Oracle Blockchain Platform.

Topics

What Are Chaincode Samples?

Oracle Blockchain Platform includes chaincode samples written in Go, Node.js, and Java to help you learn how to implement and manage your network's chaincodes.

To get to the Chaincode Samples page in the Oracle Blockchain Platform console, open the **Developer Tools** tab and select **Samples**.

The Chaincode Samples page contains:

- The Balance Transfer sample is a simple chaincode representing two parties with account balances and operations to query the balances and transfer funds between parties.
- The Marbles sample includes a chaincode to create marbles where each marble has a color and size attribute. You can assign a marble to an owner and enable operations to query status and trade marbles by name or color between owners.
- The Car Dealer 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.
  In this sample, a large auto maker and its dealers and buyers have created a blockchain network to streamline its supply chain activities. Blockchain helps them reduce the time required to reconcile issues with the vehicle and parts audit trail.
- The Fiat Money Token sample includes a chaincode to manage the complete life cycle of a fractional fungible token that represents fiat money. After you initialize the token, create token user accounts, and assign the minter role, you can issue, transfer, and burn tokens. You can also track token account balances and transaction history. For more information about the token samples, see Working With the Sample Token Specification Files.

Use the **Download sample here** links under each sample to download the sample chaincode. The download contains the Go, Node.js, and Java versions of the chaincode.

Explore Oracle Blockchain Platform Using Samples

You can install, deploy, and invoke the sample chaincodes that are included in Oracle Blockchain Platform.

You must be an administrator to install and deploy sample chaincodes. If you've got user permissions, then you can invoke sample chaincodes.

1. Go to the console and select the **Developer Tools** tab.

2. Click the **Samples** pane.

   The Chaincode Samples page is displayed.

3. Locate the sample chaincode and install it.

   a. Choose the sample chaincode that you want to use and click the corresponding **Install** button.

   b. In the Install Chaincode dialog, specify one or more peers to install the chaincode on, and select which chaincode language you want to use (Go, Node.js, or Java). Click **Install**.

4. Deploy the chaincode.

   a. Click the chaincode's **Deploy** button.

   b. In the Deploy Chaincode dialog select the channel you want to deploy the chaincode on. Click **Deploy**.

5. Go to the Channels tab and click the name of the channel that you deployed the sample chaincode on.

   a. In the Channel Information page, click the Deployed Chaincodes pane to confirm the chaincode's deployment on the channel.

   b. You can use the Ledger pane to locate information about individual transactions on the channel.

6. Click the Ledger pane and confirm the following.

   - The Ledger Summary indicates one deployment occurred. A deployment consists of an approval and a commit.
   - In the Ledger table, locate the two blocks with a Type of `data`.
   - Click the first block and in the Transactions table, click the arrow icon to display more information about the block. Confirm that the Function Name field displays `ApproveChaincodeDefinitionForMyOrg`.
   - Click the second block and confirm that the Function Name field displays `CommitChaincodeDefinition`.

7. If needed, go to the Chaincodes tab and deploy the chaincode on other channels.

   If you're working on a network that contains multiple members and have deployed the chaincode on the founder, then you don't have to deploy the chaincode on the participants where you installed the same chaincode. In such cases, the chaincode is already deployed on the participants.

   a. Locate the package ID of the chaincode you want to deploy in the table and click it.

      The Installed Peers Summary page is displayed.

   b. Click **Deployed on Channels**.

   c. On the Deployed Channels Summary page, click the **Deploy** button.

   d. In the Deploy Chaincode dialog specify the required information, and then click **Deploy**.

8. Invoke the chaincode.

   a. Go to the Chaincode Samples page, locate the chaincode you're working with, and click its **Invoke** button.

   b. In the Invoke Chaincode dialog, select a channel to run the transaction on.

   c. In the **Action** field, specify an action to complete using the chaincode.

d. Click **Execute**.

9. Confirm whether the chaincode invoked successfully.
   a. Go to the Channels tab, and locate and click the channel the chaincode was installed on.
   b. In the Ledger Summary table, locate the block number that indicates an invocation occurred.
   c. Click the block and confirm that in the Transactions table you see `Success` in the Status column.
10. If needed, go to the Samples page and invoke any other operations on the chaincode.

3

Manage the Organization and Network

This topic contains information to help you understand the console and how you can use it to manage the channels and nodes that make up your organization and the blockchain network.

Topics

What's the Console?

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

When you provisioned your Oracle Blockchain Platform instance, all of the capabilities you need to begin work on your blockchain network were added to the console.

You can use the console to perform tasks such as managing nodes, configuring network channels and policies, and deploying chaincodes. You can also monitor and troubleshoot the network, view node status, view ledger blocks, and find and view log files.

In most cases, each member of your network has its own console that they use to manage their organization and monitor the blockchain network. Your role in the network (founder or participant) determines the tasks you can perform in your console. For example, if you're a participant, you can’t add another participant to the network. Only the founder can add a participant to the network.

Also, what you can do in the console is determined by your access privileges (either Administrator or User). For example, only an Administrator can set an anchor peer or create a channel.

Your instance includes sample chaincodes that you can use to get started. See Explore Oracle Blockchain Platform Using Samples.

If the Oracle Blockchain Platform console isn't behaving as expected, then check that you're using the latest version of a supported browser. Oracle Blockchain Platform supports the following browsers:

- Mozilla Firefox
- Google Chrome
- Safari
- Microsoft Edge / Internet Explorer

The console is divided into pages.

Dashboard Page

Use the Dashboard page for an overview of the network's performance. See What Type of Information Is on the Dashboard?

On the Dashboard page, you'll find:

- A banner showing you how many different components are on your network. For example, how many channels and chaincodes.
- The number of user transactions on a channel during a specific time range.
- The number of nodes that are running or stopped.
- The number of endorsements and commits by peers.

Network Page

The Network page is where you view a list of the members in your network. The first time you use the Network page after setting up your instance, you'll see the nodes you created during set up.

You can use the Network page to:

- Find the organization IDs of the members in your network, their Membership Service Provider (MSP) IDs, and roles.
- Add a participant to the network.
- See a graphical representation of the network's structure.
- Configure, view, or import the orderer settings.
- Manage certificates.
- Add new orderering service node into network.
- Export the network config block.

Nodes Page

Go to the Nodes page to view a list of the nodes in your network. The first time you use the Nodes page after setting up your instance, you'll see:

- The console node.
- The number of peer nodes you requested when provisioning.
- The number of orderer nodes associated with your instance type. Standard has three orderer nodes and cannot be scaled up, while Enterprise has three and additional can be added.
- One Fabric certificate authority (CA) node representing the membership service.
- One REST proxy node.

During the founder instance provisioning a default channel was created and all peers were added to it.

Use the Nodes page to:

- View and set node configurations.
- Export and import peers.
- Start, stop, and restart nodes.

- Configure and start a new orderer node.
- See a graphical representation of which peer nodes are using which channels.
- Click a node's name to find more information about it.

Channels Page

The Channels page shows you the channels in your network, the peers using the channels, and the chaincodes deployed on the channels. The first time you use the Channels page after setting up your instance, you'll see the default channel that was created and all of the peers in your network added to it.

Use the Channels page to:

- Add new channels.
- See the number of chaincodes deployed on a channel.
- Click a channel's name to find more information about it, such as its ledger summary, the peers and OSNs joined to the channel, and the channel's policies and ACLs.
- Join peers to the channel.
- Manage the ordering service of the channel.
- Add or remove an ordering service node (OSN) for a channel.
- View and update the ordering service's settings.
- Configure rich history for the channel.
- Check the rich history status for the channel.
- Run and analyze rich queries on chaincodes in the channel.
- Upgrade a chaincode.

Chaincodes Page

Note that Oracle Blockchain Platform refers to smart contracts as chaincodes.

Go to the Chaincodes page to view a list of the chaincode packages installed on the instance. The first time you use the Chaincodes page after setting up your instance, no chaincodes are displayed in the list because no chaincodes were included during set up. You must add the needed chaincodes. You can add chaincodes written in Go, Node.js, and Java. You can also an external chaincode (chaincode as a service).

You can use the Chaincodes page to:

- Install and deploy a chaincode using the Quick or Advanced deploy option.
- See how many peers have a chaincode installed.
- Find out how many channels a chaincode was deployed on.

Developer Tools Page

The Developer Tools page is designed to help you learn blockchain fundamentals like how to write chaincodes and create blockchain applications.

You can use the Developer Tools page to:

- Download Blockchain App Builder for Oracle Blockchain Platform - a set of tools and samples to help you create, test, and debug chaincode projects using a command line

interface or a Visual Studio Code extension. For more information, see Blockchain App Builder for Oracle Blockchain Platform.

- Find templates and the Hyperledger Fabric mock shim to help you create chaincodes.
- Link to the SDKs and APIs that you need to write blockchain applications.
- Use the sample chaincodes to learn about chaincodes. Install, deploy, and invoke the sample chaincodes.

Modify the Console Timeout Setting

The Oracle Blockchain Platform console attempts to contact the nodes on the network for 600 seconds before it times out.

In most cases you won't have to adjust this setting, but if the console is frequently not responding, then consider increasing the timeout value. Oracle doesn't recommend decreasing the timeout value.

1. Go to the console and select the Nodes tab.
2. In the Nodes tab go to the nodes table and locate the console node. Use the nodes table's type column to find the Console node.
3. Click the node's **More Actions** menu and then click **Edit Configuration**.

   The Configure dialog is displayed.
4. In the **Request Timeout (s)** field, type or use the arrow buttons to indicate the timeout length in seconds.
5. Click **Submit**.

   The timeout length changes immediately, and you don't have to restart the console.

Monitor the Network

The console provides several ways for you to monitor the activity and health of your blockchain network. For example, you can find summary information about the total number of blocks submitted to the ledger, or you can search for and locate information about specific chaincode transactions that happened on a specific channel.

How Can I Monitor the Blockchain Network?

You can use the console to locate the following sources of information to help you understand what's happening on your network.

Network Overview Information

Use the Dashboard tab if you need at-a-glance information about how well the whole network is working and to spot any general issues such as a high rate of failing transactions. See View Network Activity .

Ledger Summary

For information about the runtime statistics for transactions on a specific channel, go to the channel's Ledger Summary area. You can drill into a specific transaction for more information about it, such as which member initiated the transaction and which peer endorsed it. See View a Channel's Ledger Activity.

What Type of Information Is on the Dashboard?

The console's Dashboard pane provides an overview of how well your network is functioning. You can use this information to identify any issue and to navigate to other panes in the console where you can learn more about and resolve any issues.

Summary Bar

This section shows the components in your network (for example, how many nodes and chaincodes). You can click a component number to go to the console tab for more information or to complete tasks related to the component.

The console displays the type of instance that you're working with. If you're a network founder, then you'll see `(Founder)`. If you're a participant in a network, then the console displays the name of the network that you're joined to. For example, `(Participant of <foundername>)`.

Health

This section shows how many nodes are running and how many are stopped in the network. Click the node numbers to go to the Nodes pane to investigate why a node might be stopped, or for more information about the nodes in the network.

Channel Activity

This area shows how many blocks have been created and how many transactions have been completed based on the number of blocks created. You might see more blocks created than user transactions. For example, if you create a channel or you deploy a chaincode, those operations are classified as system-level transactions and are included in blocks, but not classified as user transactions. This area shows the top four channels that have handled the most transactions, and for each channel shows the number of transactions that have succeeded and failed.

- User transactions are transactions that were invoked as part of the chaincode's execution, and not underlying actions such as setting up the network, creating channels, and installing and deploying chaincodes.
- A block can contain multiple user transactions.

You can filter the amount of activity information that is displayed. You can select a set time range (for example, last hour or last week), or you can select **Custom** and pick the dates that you want activity information for.

Peer Activity

This area shows the number of endorsement and commits completed by the network's peer nodes. This area shows the top four peer nodes that have endorsed and committed the most transactions, and for each of those four peers, this area shows the number of endorsements and commits that have succeeded and failed.

- A transaction is an endorsement, and a commit is when a transaction is written to the block.
- Commits can be either user transactions or system transactions.
- Commits are the number of transactions that have been committed to the block. Commits aren't blocks.
- Only specific peers must do endorsements, but all peers must do commits.

You can filter the amount of activity information that is displayed. You can select a set time range (for example, last hour or last week), or you can select **Custom** and pick the dates that you want activity information for.

View Network Activity

Use the console's Dashboard tab to find information about your blockchain network's activities, such as percentage of nodes that are running or stopped, and how successfully the network is executing chaincode transactions.

You can use this information as a starting place and then use the other tabs in the console to drill into any issues that you discover. For information about what displays in the Dashboard tab, see What Type of Information Is on the Dashboard?

1. Go to the console and select the Dashboard tab.
2. To see channel and peer activity information that occurred at a specific time such as for the last week or month, go to the filter dropdown menu and select the time range you want. Select Custom to enter specific begin and end dates and click **Apply**.

Manage Nodes

This topic contains general information about managing the nodes in your network, such as describing the types of nodes in your blockchain network, how to view your nodes and their topology, how to stop and start them, and how to set logging levels for a node.

What Types of Nodes Are in a Network?

A blockchain network contain console, peer, orderer, certification authority (CA), and REST proxy nodes. The nodes that display in your console depend upon if you're the founder of or a participant in a network.

For example, if you're a participant in a network, your console won't display an orderer node for that network. If you're a founder, your console displays all node types.

What nodes are included in a new instance?

After you provision your instance and access the Nodes tab for the first time, you'll see:

- One console node.
- The number of peers you requested during set up. These peers display with the Peer(Member) type. The maximum number of peer nodes that can be included with an instance is 16.
- An orderer node, or ordering service node (OSN), representing an ordering service.
- A Fabric certificate authority (CA) representing the membership service.
- A REST proxy node.

I need more information about the different node types

Use this table to find more information about nodes.

Node Type	What Does This Node Do?	Displays In Founder or Participant Instance	Number of Nodes per Instance	Can I Add Another Node After Provisioning My Instance?
CA	This node provides and manages peer node credentials and member credentials.	Founder Participant	1	No
Console	This node is the console component.	Founder Participant	1	No
Orderer	This node provides communication between nodes. It guarantees the delivery of transactions into blocks and blocks into the blockchain. If you're a participant, then you must import the founder's ordering service setting into your instance so that all peer nodes can communicate.	Founder Participant	3	Digital Assets Edition: Yes Enterprise Edition: Yes Standard Edition: No
Peer	This node contains a copy of the ledger and writes transactions to the ledger. This node can also endorse transactions. Your network can contain member or remote peers.	Founder Participant	2 to 16 The number of peer nodes you can add was specified when your instance was created.	Yes
REST Proxy	This node maps an application identity to a blockchain member, which allows users and applications to call the Oracle Blockchain Platform REST APIs.	Founder Participant	1	No

Find Information About Nodes

This section contains information about where in the console you can find information about the nodes in your instance and network.

View General Information About Nodes

Use the Nodes tab to view general information about all of the nodes in your network. For example, Name, Route, Type, and Status.

You can also use the Nodes tab to drill into details about a specific node. For more information about node types, see What Types of Nodes Are in a Network?

1. Go to the console and select the Nodes tab.
2. In the Nodes tab confirm that the **List View** (and not the **Topology View**) is displaying.

Column	Description
Route	Oracle Blockchain Platform generated the URLs when you provisioned your instance or when you create nodes. If you use the Hyperledger Fabric SDK, you need these URLs to specify which peers you want the SDK to interact with.
Type	Indicates the node type.
MSP ID	Membership Service Provider ID.
Status	Indicates if the node is running or down. Also indicates if there's an unapplied configuration change for the node. Note the following statuses: • **Up** — The node is running and working normally. • **Down** — The node is stopped. • **N/A** — This status displays for remote peers because your instance doesn't have the permissions required to get the peer's status.
IsConfigured	If the node's configuration was updated you need to restart the node for the updates to take effect. Nodes with the yes status are running (and not stopped).
More Actions Menu	Your permissions determine the options available from the More Actions menu. If you're an administrator, this button provides links to modify the node's configuration. Administrators and users can stop, start, and restart nodes.

Access Information About a Specific Node

Use the Nodes tab to access information about a specific node, such as health information or log files.

1. Go to the console and select the Nodes tab.
2. Click a node's name to go to the Node Information page. The panes that are displayed in the Node Information page depend on the node type that you select.

Pane	Available for which node types?	What can I do in this pane?
Health	All	For a peer node, this pane displays information about endorsed and committed transactions.
Logs	All	View and download log files to discover and troubleshoot issues with a node.
Channels	Peer	View a list of channels that the selected peer node is using for communication with other nodes. Join the peer node to other existing channels as needed. Go to the Channel page to create a channel and specify which peer nodes can join it.
Chaincodes	Peer	View the chaincodes that are installed on the peer node. Go to the Chaincode page to install a new chaincode or upgrade an existing chaincode.
Transaction Statistics	REST proxy	View the total queries, failed queries, total invocations, and failed invocations handled by the REST proxy.

View a Diagram of the Peers and Channels in the Network

Use the Topology view to access an interactive diagram that shows which network peers are using which channels.

1. Go to the console and select the Nodes tab.
2. In the Nodes tab, click **Topology View** to see a diagram showing the peer nodes in your network and which channels they're using.

3. Hover over a peer to highlight it and the channels it's using.

Find Node Configuration Settings

Use the Nodes tab to find a specific node's configuration settings. If you're an administrator, then you can update a node's configuration settings. If you're a user, then you can view a node's configuration settings.

1. Go to the console and select the Nodes tab.
2. Go to the Nodes table, locate the node that you want configuration setting information for, and click the node's **More Actions** button.
3. The configuration option is determined by your permissions. If you're an administrator, locate and click **Edit Configuration**. If you're a user, locate and click **View**.

   The Configure dialog is displayed, showing the attributes specific to the node type you selected. See Node Configuration.

Start and Stop Nodes

You can start or stop CA, peer, and REST proxy nodes in your network. You can start or restart orderer nodes. You can't start or stop the console node or remote peer nodes.

You can start and stop nodes depending upon the traffic in your network. For example, if network traffic is light, then you can stop unneeded peer nodes.

You can also restart a node. See Restart a Node.

When you stop a peer node, Oracle Blockchain Platform removes the peer's listing on the Channel page and Chaincodes page. If you stop all peers that have the chaincode installed, the Chaincodes page doesn't list the chaincode. If you stop all peers joined to a channel, the Channels page lists the channel, but its information isn't available to view.
Before stopping a node for an extended period of time, transfer all this peer's responsibilities to other running peers, and then remove all the responsibilities this peer has.

- Check all other peers' gossip bootstrap address lists, remove the peer address, and add another running peer's address if needed. After peer configuration change, restart the peer.
- Check all channels' anchor peer lists, remove the peer from the anchor peer lists, and add another running peer to the anchor peer list if needed.
- If a channel is joined only to this peer, or if chaincode is deployed only on this peer, consider using another running peer to join the same channel and deploy the same chaincode.

You must be an administrator to complete this task.

1. Go to the console and click the **Nodes** tab.
2. On the Nodes page, go to the Nodes table, locate the node that you want to start or stop, and click the node's **More Actions** button.
3. Click either the **Start** or **Stop** option. The node's status changes to either up or down and information is written to the node's log file.

Restart a Node

You can restart the CA, orderer, peer, and REST proxy nodes in your network. You can't restart the console node or remote peer nodes.

Restart a node if it's not responding or running properly, or if you've updated a node's configuration. You can also start or stop a node. See Start and Stop Nodes.

You must be an administrator to complete this task.

1. Go to the console and click the Nodes tab.
2. On the Nodes page, go to the Nodes table, locate the node that you want to restart, and click the node's **More Actions** button.
3. Click **Restart**.

   The node's status changes to `restarting` and information is written to the log file.

Set the Log Level for a Node

If you're an administrator, then you can specify the type of information you want to include in a node's log files. For example, ERROR, WARNING, INFO, or DEBUG.

By default, every node's log level is set to INFO. When developing and testing your network, Oracle suggests that you set the logging level to DEBUG. If you're working in a production environment, then use ERROR.

Only an administrator can change a node's log level setting. If you're a user, then you can view a node's log level settings.

1. Go to the console and select the Nodes tab.
2. In the Nodes tab, go to the nodes table, locate the node you want to update, click its **More Actions** menu, and click **Edit Configuration**.

   If you have user permissions, then your console will have the **View** option that you click to see the node's log level setting and other configuration settings.

   The Configure dialog is displayed.
3. In the **Log Level** field, select the log level you want to use.
4. Click **Submit**.

Manage Channels

This topic contains information about managing the channels in your network, such as how to create and view channels, how to join peers and designate and anchor peer, how to work with policies and access control lists, and how to associate orderers with a channel.

What Are Channels?

Channels partition and isolate peers and ledger data to provide private and confidential transactions on the blockchain network.

Members define and structure channels to allow specific peers to conduct private and confidential transactions that other members on the same blockchain network can't see or access. Each channel includes:

- Peers
- A shared ledger
- Chaincodes deployed on the channel
- One or more ordering service nodes
- Channel policy definitions and ACLs where the definitions are applied

Each peer that joins a channel has its own identity that authenticates it to the channel peers and services. Although peers can belong to multiple channels, the information on transactions, ledger state, and channel membership is restricted to peers within each channel.

You can use the Oracle Blockchain Platform console or the Hyperledger Fabric SDK to create channels on your blockchain network. See View Channels.

View Channels

Members in your network use channels to privately communicate blockchain transactions information.

Use the Channel page to view a list of the channels in your network, create and monitor channels, specify anchor peers, and upgrade the deployed chaincodes used on your channels.

1. Go to the console and click the **Channels** tab.

   The Channels page is displayed. The channel table contains a list of all of the channels on your network.

2. In the channel table, click the channel name that you want information about. If all peers joined to the channel are stopped, the channel is listed but its information isn't available to view.

   The Channel Information page is displayed.

3. Click through the Channel Information page's panes to find information about the channel.

Section	What can I do in this pane?
Ledger	Get information about the channel's ledger activity such as block number and the number of user transactions in the block. Click a block number to drill into information about its transactions. You can use the filter field to specify the summary information that you want to see (for example, information from the last day or last month), or use the custom option to enter start and end times. See View a Channel's Ledger Activity.
Deployed Chaincodes	View the list of chaincodes that have been deployed on the channel.
Orderers	View a list of the orderers currently active, and add a new OSN to join the channel.
Peers	View the list of peers that are joined to the channel. Use this section to set anchor peers for the channel.
Organizations	View the list of network members whose peers are using the channel to communicate.
Channel Policies	View the list of the standard policies and any policies that you created for the channel. Use this section to add, modify, and delete policies.
ACLs	View the access control lists (ACLs) and the policies used to manage which organizations and roles can access the channel's resources.

Create a Channel

You can add channels to the network and specify which members can use the channel, and which peers can join the channel. You can't delete channels.

You must be an administrator to complete this task.

1. Go to the console and select the Channels tab.
2. In the Channels tab, click **Create a New Channel**.
3. In the **Channel Name** field, enter a unique name for the channel. The channel's name can be up to 128 characters long.
4. In the Organizations section, select any additional members that you want to communicate on the channel.

   If you're working in a participant instance, you need to add the founder to your instance before the founder's MSP ID displays in the Organization section. To add the founder organization, go to the Network tab and click the **Add Organization** button to upload the founder's certificates.
5. In the **MSP ID ACL** section, specify the organizations that have access to the channel and permissions for each selected organization. Note that you can add more organizations to or delete them from the channel later, as needed.

   Your organization's permissions are set to write (ReaderWriter) and you can't modify this setting. By default, other member's permissions are set to write (ReaderWriter), but you can change them to read (ReaderOnly) if you don't want the members to invoke chaincodes and to only read channel information and blocks on the channel.
6. (Optional) In the **Peers to Join Channel** field, select one or more peers. Note the following information:
   - If your network contains participants, the participants' peers don't display in this list. Participants must use their consoles to join peers to the channel. A participant can't join its peers to the channel unless its organization was added to the channel's MSP ID ACL section.
   - If you want to create the channel only, then don't select any peers. You can add peers to the channel later.
7. Click **Submit**.

   The channel table displays the new channel.

After you create the channel, you can complete the following tasks:

- Deploy a chaincode on the channel. See Deploy a Chaincode.
- If the network contains participants, then those participants use their consoles to join member peers to the channel. See Join a Peer to a Channel.

View a Channel's Ledger Activity

Use the ledger to find summary information and runtime statistics for transactions on a specific channel.

1. Go to the console and select the Channels tab.
2. In the channel table, click the channel name that you want transaction information about. In the Channel Information page, confirm that the Ledger pane is selected.

3. Use the Ledger Summary area to find basic information about the channel's activity, such as the total number of blocks in the ledger's chain and the total number of user transactions on the channel.
4. To see blockchain activity that occurred at a specific time such as the last day or week, use the filter drop-down list to select the time range that you want. To locate and drill down into a specific set of transactions, select **Custom** and enter search criteria in the **Start Time** and **End Time** fields, or click the calendar icon and pick the dates that you want. Click **Apply**.

   If you select a specific time period (for example, **Last day**) and then select it again to re-run the query, the query doesn't run again. To get the latest information, click the **Refresh** button.

   The following transaction types can be displayed for a block:

   - genesis — The transaction that runs the configuration block to initialize the channel.
   - data (sys) — The transaction that starts the chaincode's container to make the chaincode available for use.
   - data — A chaincode transaction called for execution on the channel.
5. To find more information about a specific transaction, locate the transaction in the query ledger table and click it. The transactions table displays the transaction's details.

   For any given block, the transactions in the table are listed in the order of the transaction number, which is assigned by the ordering service when the block is created. Because of this, the transactions listed in the table might have time stamps (which are from the peer's endorsement of the chaincode) that are before or after other transactions in the same block. The time range of transactions in a single block is governed by ordering service settings including the batch timeout parameter (the time that the ordering service waits for additional transactions after an initial transaction before cutting a block).

Transaction Detail	Description
TxID	The unique alphanumeric ID assigned to the transaction. The TxID is constructed as a hash of a nonce concatenated with the signing identity's serialized bytes.
Time	The transaction's time stamp (date and time that the transaction occurred).
Chaincode	The name of the chaincode that ran the transaction. This field can show the name of a chaincode that you wrote, installed, and deployed, but can also show a system chaincode. System chaincode options are: • _lifecycle — For lifecycle requests, such as install, deploy, and upgrade. • QSCC — For querying. This chaincode includes APIs for ledger query.
Status	Status that indicates whether the transaction succeeded or failed.

6. Click the triangle icon next to the TxID to view in-depth information about the transaction, such as function name, arguments, validation results, response status, the initiator, and the endorser.

   If a transaction failed, you can use the TxID to search the error logs on the peer node or orderer nodes for more information.

View or Update a Channel's Organizations List

You can view the list of the organizations that have access to the channel. If you created the channel, then you can change an organization's permissions on the channel, and you can add organizations to or remove them from the channel

1. Go to the console and click the **Channels** tab.

   The Channels page is displayed. The channel table contains a list of all of the channels in your network.

2. In the channels table, locate the channel that you want information about, click the channel's **More Actions** button, and then click **Edit Channel Organizations**.

   The Edit Organizations page is displayed.

3. In the **MSP ID ACL** section, you can complete the following tasks:

   - Modify an organization's permissions. The organization that created the channel is set to write (`ReaderWriter`). You can't change this setting.
   - If you're the network founder, you can clear an organization's checkbox to delete it from the channel. If you're a network participant, click the **Delete** button to delete an organization from the channel. If you delete an organization from a channel, the organization and its peers can no longer query, call, or deploy a chaincode on the channel. The removed organization's peers can't join the channel.
   - Click an organization's checkbox to add the organization to the channel and set its permissions. By default, each member's permissions is set to write (`ReaderWriter`), but you can change it to read (`ReaderOnly`) if you don't want the member to call chaincodes and to only read channel information and blocks on the channel.

4. Click **Submit** to save the changes.

Join a Peer to a Channel

You can add a peer node to a channel so that the node can use it to exchange private transaction information with other peer nodes on the channel.

Note the following information:

- When you create a channel, you specify which local peer nodes can join the channel.
- If you're creating a network containing a participant, then you can select the participant as a member on the channel. Or you can add the participant after the channel is created.
- Your instance has multiple availability domains or fault domains, and Oracle recommends that you join one peer from each partition to the channel. This is because if one worker node is unavailable that the channel is still available for endorsements and commits. To determine which domain a peer is located in, in the **More Actions** menu select **Show AD Info** to see the availability domain information.
- You can join a maximum of seven peers from each domain.

See Create a Channel.

You must be an administrator to perform this task.

1. Go to the console and select the Nodes tab.
2. In the Nodes tab, click the peer node that you want to add to a channel.

3. In the Node Information page, click the Channels pane to view the list of channels the peer is already using.
4. Click **Join New Channels**.

   The Join New Channels dialog is displayed.
5. Click the **Channel Name** field and from the list select the name of the channel to join. Click the field again to select another channel. Click **Join**.

Add an Anchor Peer

Each member using a channel must designate at least one anchor peer. Anchor peers are primary network contact points, and are used to discover and communicate with other network peers on the channel.

You can designate one or more peers in your organization as an anchor peer on a channel. For a high availability network, you can specify two or more anchor peers. All members using the network channel must use their console to designate one or more of their peer nodes as anchor peers.

You must be an administrator to perform this task.

1. Go to the console and select the Channels tab.

   The Channels tab is displayed and the channel table contains a list of all of the channels on your network.
2. In the channels table, click the channel name you want to add anchor peers to.

   The Channel Information page is displayed.
3. In the Channel Information page, click the Peers pane.
4. Locate the peer or peers that you want to designate as anchor peers and click their **Anchor Peer** checkboxes to select them.
5. Click the **Apply** button.

Change or Remove an Anchor Peer

You can change or remove a channel's anchor peers. Anchor peers are primary network contact points, and are used to discover and communicate with other network peers on the channel.

Before you change or remove the channel's anchor peers, note the following information:

- To communicate on the channel, you must designate one or more peers in your organization as an anchor peer.
- For a high availability network, you can specify two or more anchor peers.
- All members using the network channel must use their console to designate one or more of their peer nodes as anchor peers.

You must be an administrator to perform this task.

1. Go to the console and select the Channels tab.

   The Channels tab is displayed and the channel table contains a list of all of the channels on your network.
2. In the channels table, click the channel name you want to remove anchor peers from.

   The Channel Information page is displayed.

3. In the Channel Information page, click the Peers pane.
4. Locate the peer or peers that you want to remove as anchor peers and clear their **Anchor Peer** checkboxes. Alternatively, to add another peer as an anchor peer, click its **Anchor Peer** checkbox to select it.
5. Click the **Apply** button.

View Information About Deployed Chaincodes

You can view information about the chaincodes that are deployed on the different channels in your network.

You might need information about deployed chaincodes to determine if you need to upgrade the chaincode, or to find out which channels the chaincode was deployed on.

1. Go to the console and select the Channels tab.
2. In the channels table, click the channel name with the chaincode that you want to view information for.
3. In the Channel Information page, confirm that the Deployed Chaincodes pane is selected
4. In the chaincode table, you can:
   - Click the chaincode package ID to go to the Chaincodes tab to learn more information about it. For example, the peers that the chaincode is installed on and the channels that the chaincode is deployed on.
   - In a chaincode's More Actions menu, click **View Chaincode Definition** to find details about the chaincode's definition, including the endorsement policy.
5. (Optional) If you see a channel listing without a chaincode, then you can go to the Chaincodes tab and deploy a chaincode to the channel. See Deploy a Chaincode.

Work With Channel Policies and ACLs

This topic contains information about a channel's policies and access control lists (ACLs). It provides an overview of what policies are, policy types, and how to modify them, as well as how to use ACLs to manage which organizations and roles can access a channel's resources.

What Are Channel Policies?

A policy defines a set of conditions. The required parties must meet the policy's conditions before their signatures are considered valid and the corresponding request happens on the network.

The blockchain network is managed by these policies. Policies check the identity associated with a request against the policy associated with the resource needed to fulfill the request. Policies are located in the channel's configuration.

After you configure the channel's policies, you assign them to the channel's ACLs resources to determine which members are required to sign before a change or action can happen on the channel. For example, suppose you modified the Writers policy to include members from Organization A or Organization B. Then you assigned the Writers policy to the channel's `cscc/GetConfigBlock` ACL resource. Now only a member from Organization A or Organization B can call `GetConfigBlock` on the `cscc` component.

What Are the Policy Types?

There are two policy types: Signature and ImplicitMeta.

- **Signature** — Specifies a combination of evaluation rules. It supports combinations of *AND*, *OR*, and *NOutOf*. For example, you could define something like “An admin of org A and 2 other admins" or "11 of 20 org admins.”
  Any new policies you create will be Signature policies.
- **ImplicitMeta** — This policy type is only valid in the context of configuration. It aggregates the result of evaluating policies deeper in the configuration hierarchy, which are defined by Signature policies. It supports default rules, for example “A majority of the organization admin policies.”

When Are Policies Created?

When you add a channel to the network, Oracle Blockchain Platform creates default policies. The default policies are: Admins, Writers, Readers, Endorsement, LifecycleEndorsement (ImplicitMeta policies), and Creator (Signature policy). You can modify these policies as needed or create policies.

Note the following important issue about channel policies:

- You can use the console to create a channel and set your organization's ACL to ReaderOnly. After you save the new channel, you can't update this ACL setting from the channel's Edit Organization option.

  However, you can use the console's Manage Channel Policies functionality to add your organization to the Writers policy, which overwrites the channel's ReaderOnly ACL setting.

Add or Modify a Channel's Policies

You can add or modify a channel's policy to specify which members are required to perform a specific action on the channel. After you define policies, you assign them to the channel's ACLs.

Before you add or update policies, you need to understand how Oracle Blockchain Platform creates default channel policies. See What Are Channel Policies?.

You must be an administrator to perform this task.

1. Go to the console and select the Channels tab.
   The Channels tab is displayed and the channel table contains a list of all of the channels on your network.
2. In the channels table, click the channel name that you want to add policies to or modify policies for.
   The Channel Information page is displayed.
3. In the Channel Information page, click the Channel Policies pane.
4. Do one of the following:
   - To add a new policy, click the **Create a New Policy** button. The **Create Policy** dialog is displayed. Enter a name in the **Policy Name** field and select Signature in the **Policy Type** field. Expand the **Signature Policy** section.
   - To modify an existing policy, click a policy's name. The **Update Policy** dialog is displayed.
5. Click the **Add Identity** button to add an organization. Or modify an existing signature policy as needed. Note the following information:

Field	Description
MSP ID	From the dropdown menu, select the organization that must sign the policy.
Role	Select the corresponding peer role required by the policy. Usually this will be member. You can find a peer's role by viewing its configuration information.
Policy Expression Mode	In most cases, you'll use **Basic**. Select **Advanced** to write an expression string using *AND*, *OR*, and *NOutOf*. For information about how to write a valid policy expression string, see Endorsement policies in the Hyperledger Fabric documentation.
Signed By	Select how many members must sign the policy to fulfill the request.

6. If you're adding a new policy, then click **Create**. If you're modifying a policy, then click **Update**.

Delete a Channel's Policies

You can delete channel policies that you have created.

You can't delete the default policies: Admins, Creator, Readers, Writers, Endorsement, and LifecycleEndorsement. Also, you can't delete a channel policy if it is assigned to an ACL. Before you try to delete a channel policy, confirm that the policy isn't assigned.

You must be an administrator to perform this task.

1. Go to the console and select the Channels tab.
   The Channels tab is displayed and the channel table contains a list of all of the channels on your network.
2. In the channels table, click the channel that you want to delete a policy from.
   The Channel Information page is displayed.
3. In the Channel Information page, click the Channel Policies pane.
4. Locate the policy that you want to delete and click its **More Options** button.
5. Click **Remove** and confirm the deletion.

What Are Channel ACLs?

Access control lists (ACLs) use policies to manage which organizations and roles can access a channel's resources.

Users interact with the blockchain network by targeting components such as the query system chaincode (`qscc`), lifecycle system chaincode (`_lifecycle`), configuration system chaincode (`cscc`), peer, and event. These components are associated with specific resources (for example, `GetConfigBlock` or `GetChaincodeData`) that you can assign policies to at the channel level. These policies are a part of the channel's configuration.

A policy defines which organizations and roles can request a resource. When a request is made, the policy tells the system to check the requester's identity and determine if it's authorized to make the request. When you create a channel, Oracle Blockchain Platform includes the default Hyperledger Fabric ACLs with the channel. Oracle Blockchain Platform also creates default policies (Admin, Creator, Writers, Readers, Endorsement, and

LifecycleEndorsement) for the channel. You can modify these policies or create policies as needed. See What Are Channel Policies?.

Update Channel ACLs

You can update the channel's ACLs by assigning policies to the channel's resources. A policy defines which organizations and roles can request a resource

You must understand what policies and ACLs are before you update a channel's ACLs. For more information, see What Are Channel ACLs?.

1. Go to the console and click the **Channels** tab.
   The Channels page is displayed and the channel table contains a list of all of the channels on your network.
2. In the channels table, click the name of the channel that you want to update ACLs for.
   The Channel Information page is displayed.
3. In the Channel Information page, click the ACLs pane.
4. In the Resources table, locate the resource that you want to update. Click the resource's **Expand** button and select the policy that you want to assign to the resource.
5. Modify the other resource's policies as needed.
6. Click **Update ACLs**.

Add or Remove Orderers To or From a Channel

The orderer admin organization can add or remove orderers from a channel.

To add orderers to a channel:

1. In the founder console, open the Channels tab and select the channel to see its details view.
2. Open the Orderers subtab. All orderer nodes currently joined to the channel are listed.
3. Click **Join Channel**. Select an OSN not yet in this channel and click **Join**.

To remove orderers from a channel:

1. In the founder console, open the Channels tab and select the channel to see its details view.
2. Open the Orderers subtab. All orderer nodes currently joined to the channel are listed.
3. Select the orderer you want to remove from the channel and from its More Actions menu select **Remove**.

Set the Orderer Administrator Organization

You can assign the administration of OSNs in a channel to any organization. Normally either the founder or the channel creator would be assigned.

1. In the founder console, open the Channels tab.
2. Select the channel for which you want to set the orderer administrator organization, and from the Action menu select **Manage OSNs Admin**.
3. Select from the list of available organizations, and click **Submit**.

Edit Ordering Service Settings for a Channel

You can update the ordering service settings for a particular channel.

- You can update the ordering service settings for the entire network as described in Edit Ordering Service Settings for the Network.
- If you change the ordering service settings and there are applications running against the network, then those applications must be manually updated to use the revised ordering service settings.
- It isn't common, but in some situations, you might expose a different ordering service to some of the network participants. In this case, you'll export the updated network configuration block and the required participants will import the revised settings. See Join the Participant or Scaled-Out OSNs to the Founder's Ordering Service.

You must be an administrator to complete this task.

1. Go to the founder's console and click the **Channels** tab.
2. Locate the channel, click the **More Actions** menu, and select **Update Ordering Service Settings**.

   The Ordering Service Settings dialog box is displayed.
3. Update the settings as needed.

Field	Description
Batch Timeout (ms)	Specify the amount of time in milliseconds that the system will wait before creating a batch. Enter a number between 1 and 3600000.
Max Message Count	Specify the maximum number of messages to include in a batch. Enter a number between 1 and 4294967295.
Absolute Message Bytes	Specify the maximum number of bytes allowed for the serialized messages in a batch. This number must be larger than the value you enter in the Preferred Message Bytes field.
Preferred Message Bytes	Specify the preferred number of bytes allowed for the serialized messages in a batch. A message larger than this size results in a larger batch, but the batch size will be equal to or less than the number of bytes you specified in the Absolute Message Bytes field. Typically you set this value to 1 MB or less. The value that you enter in this field must be smaller than the value you enter in the Absolute Message Bytes field.
Snapshot Interval Size	Defines number of MB at which a snapshot is taken.

4. Click **Update**.

   The updated settings are saved.

Manage Certificates

This topic contains information about how to manage your network's certificates, including how to import and export certificates to set up your blockchain network, and how to manage and revoke certificates.

Typical Workflows to Manage Certificates

Here are the common tasks for managing your network's certificates.

Adding Organizations to the Network

You must be an administrator to perform these tasks.

Task	Description	More Information
Export or prepare an organization's certificates	The organization that wants to join the network either outputs or writes its certificates file and gives it to the founder.	Export Certificates Create an Organization's Third-Party Certificates File
Import member certificates	The founder imports the organization's certificates file to add the organization to the network.	Import Certificates to Add Organizations to the Network
View certificates	The founder can view and manage the network's certificates.	View and Manage Certificates

Revoking Certificates

You must be an administrator to perform these tasks.

Task	Description	More Information
Decide which certificates to revoke	View the certificates on your system to determine which ones to revoke to keep the network secure.	View and Manage Certificates
Select the certificates to revoke	Revoke the certificates in your CA.	Revoke Certificates
Apply CRL	Generates and applies an updated CRL to ensure that clients with revoked certificates can't access channels.	Apply the CRL

Export Certificates

Founders and participant organizations must import and export certificate JSON files to create the network.

Note the following information:

- For the founder to add a participant organization to the blockchain network, the participant must export its certificates file and make it available to the founder. The founder then uploads the certificates file to add the participant organization to the network.

- The certificate export file contains admincerts, cacerts, and tlscacerts.
- You might need to export certificates for blockchain or application developers. For example, a client application needs the TLS certificate to interact with peers or orderers.

For information about writing certificate files required to add Hyperledger Fabric or Third-Party organizations to the network, see Extend the Network.

1. Go to the console and select the Network tab.
2. In the Network tab, go to the Organizations table, locate the member that you want to export certificates for, and click its **More Actions** button.
3. Click **Export Certificates**.

   Note that files exported by the console and REST APIs are only compatible for import with the same component. That is you can't successfully use the REST API to import an export file created with the console. Likewise, you can't successfully use the console to import an export file created with the REST API.
4. Specify where to save the file. Click **OK** to save the certificates file.
5. Send the certificates JSON file to the founder for import. See Import Certificates to Add Organizations to the Network.

Import Certificates to Add Organizations to the Network

To add an organization to the network, the founder must import a certificates file that was exported or prepared by the organization that wants to join the network.

You can import certificates for the following organization types.

Type	Description
Oracle Blockchain Platform Participant Organization	You can import a participant organization into a Oracle Blockchain Platform network. You upload the certificates that the participant organization exported from the console and sent to you. For information about creating certificates for upload and a list of the other steps that you need to perform to successfully set up a participant organization on the network, see Join the Participant or Scaled-Out OSNs to the Founder's Ordering Service.

You must be an administrator to import certificates.

1. Go to the console and select the Network tab.
2. In the Network tab, click **Add Organizations**. The Add Organizations page is displayed.

   Note that files exported by the console and REST APIs are only compatible for import with the same component. That is you can't successfully use the REST API to import an export file created with the console. Likewise, you can't successfully use the console to import an export file created with the REST API.
3. Click **Upload Organization Certificates**. The File Upload dialog is displayed.
4. Browse for and select the JSON file containing the certificate information for the organization you want to add to the network. Usually this file is named `certs.json`. Click **Open**.
5. (Optional) Click the plus (+) icon to locate and upload another organization's certificate information.

6. Click **Add**. The organizations that you added are displayed in the Organization table.

   Note the following information for Oracle Blockchain Platform participant and Hyperledger Fabric organizations. Even though the founder uploaded the certificate information, the added organization can't use the ordering service to communicate on the network until it imports the founder's ordering service settings. The founder must export its ordering service settings and give the resulting file to the joining organizations for import. See one of the following:

   - For Oracle Blockchain Platform participants, see Join the Participant or Scaled-Out OSNs to the Founder's Ordering Service.

What's a Certificate Revocation List?

You use a certificate revocation list (CRL) to help manage the certificates throughout your network.

A CRL is a list of digital certificates that the issuing Certificate Authority (CA) has revoked before their scheduled expiration date and must no longer be trusted and used on the network. For example, you use a CRL to revoke any certificates that have been lost, stolen, or compromised.

After you use the Manage Certificates functionality to revoke certificates for users, Oracle Blockchain Platform creates the CRL. To ensure that the certificates are revoked throughout the network, you'll need to complete the following task:

- Apply the CRL after you join peers to a channel created by another network member. Applying a CRL prevents clients with revoked certificates from accessing the channel. See Apply the CRL.

View and Manage Certificates

Use the console to view and manage the user certificates in your instance and any of the certificates you imported when building the network.

1. Go to the console and select the Network tab.
2. In the Network tab, locate your organization's ID and click its **More Actions** button. Select **Manage Client Certificates**.

   Note that the Certificate Summary table will be empty until you add users to your instance. Also, the administrator's certificate doesn't display in this table. This is to prevent you from accidentally revoking the administrator's certificate.

   Organizations with third-party certificates with revoked certificates won't display in this table. In such cases, you must use the native Hyperledger Fabric CLI or SDK to import the organization's certificate revocation list (CRL) file.

   The Certificates Summary dialog is displayed and shows a list of the certificates in your instance.
3. As needed, perform any of the following tasks:

   - Revoke certificates. See Revoke Certificates.
   - If you've revoked certificates and are working in a network with multiple members, then use Apply CRL after you join peers to a channel created by another network member. Apply CRL prevents clients with revoked certificates from accessing the channel. See Apply the CRL.

Revoke Certificates

An organization can revoke certificates for any of its users. To ensure that the network remains secure, revoke certificates if they're lost, stolen, or compromised.

You must be an administrator to complete this task.

1. Go to the console and click the Network tab.
2. On the Network page, locate your organization's ID and click its **More Actions** button. Select **Manage Client Certificates**.

   The Certificates Summary dialog is displayed.
3. In the Certificates Summary dialog box, locate and select the IDs of the users that you want to revoke certificates for.
4. Click **Revoke** and confirm that you want to permanently revoke certificates for the selected users.

   The users with revoked certificate display in the table and are added to the CRL.
5. If you're working in a network with other members, then to ensure that their revoked certificates are cleaned up across the network, you must do the following:
   - If you're working in a network with multiple members, then apply the CRL after you join peers to a channel created by another network member. Applying the CRL prevents clients with revoked certificates from accessing the channel. See Apply the CRL.

Apply the CRL

If you're working in a network, then you must apply the CRL after you join peers to a channel created by another network member. Apply CRL prevents members with revoked certificates from accessing the channel.

You must do the following tasks before applying the CRL:

- Revoke certificates. See Revoke Certificates

You must be an administrator to perform this task.

1. Go to the console and select the Network tab.
2. In the Network tab, locate your organization's ID and click its **More Actions** button. Select **Manage Client Certificates**.

   The Certificates Summary dialog is displayed.
3. Click the Apply CRL button and confirm that you want to apply the CRL.

Manage Ordering Service

This topic contains information about how founders and participants manage the ordering service.

In addition to the content covered in this topic, several channel-specific tasks for the orderer nodes can be performed on the Channels page of the console. See:

- Add or Remove Orderers To or From a Channel
- Set the Orderer Administrator Organization
- Edit Ordering Service Settings for a Channel

What is the Ordering Service?

Oracle Blockchain Platform supports Raft as the consensus type.

For more information on the Hyperledger Fabric implementation of the Raft protocol, see: The Ordering Service - Raft.

With the older Kafka consensus type, the whole network can have at most two orderer nodes, and they have to join all channels. In some cases, they may be overloaded, and cannot be scaled out. With the Raft consensus type, the network can have an arbitrary number of orderer nodes, and each channel can define its own orderer node set. Different channels can use different orderer nodes, and orderer nodes will no longer be the bottleneck.

However, the Raft consensus type can be complicated to configure properly. There are rules about what can or can't be done, and if these rules are not followed the channel and even the network may not work. The following guidelines can reduce the number of problems that you encounter:

Keep the Majority of the Ordering Service Nodes (OSN) Alive

The Raft consensus algorithm requires that the majority of ordering service nodes (OSNs) are working; otherwise no consensus can be made. Majority means greater than 50%. For example, for five OSNs, there must be at least three OSNs working; for six OSNs, there must be at least four OSNs working.

- If there are 50% or fewer OSNs working in the network, network management will no longer be functional. No channels can be created, no new orderer nodes can be added into network, no orderer can be removed from network, and so on.
- If there are 50% or fewer OSNs working in the application channel, no transaction can be submitted to this application channel. Queries may still function correctly, however administrative operations such as adding a new organization, changing the access control list, or deploying chaincodes will fail.

Be cautious when adding a new OSN to the network or an application channel. Ensure the owner is trustworthy and the OSN is robust.

When removing OSNs or an organization, ensure that more than 50% of the OSNs will remain working. For example, if you had two organizations with three OSNs each, if you removed one organization, during the removal it would be interpreted as only 50% of the OSNs being functional. Add an OSN to the remaining organization before deleting the extraneous organization to ensure that you always exceed 50% of the OSNs working.

Do Not Add or Remove Orderers Frequently

Every time a new OSN is added into a network or channel, or an existing OSN is removed from a channel, the current Raft OSN cluster will briefly become unstable. During this period, no transactions can be handled, and an error message similar to the following may indicate such a status:

```
UNKNOWN: Stream removed
SERVICE UNAVAILABLE
BAD REQUEST
```

This might last a few minutes. If you have removed the previous Raft leader OSN from the channel, this might last as long as 20 minutes.

Ensure that you aren't adding or removing orderers frequently. If multiple orderers must be added or removed, do one at a time and ensure that the network has returned to operational status before making the next change.

Ensure the New Orderer is Started As Soon As Possible

When adding a new orderer into the network, usually two organizations will be involved: the founder and the owner of the new orderer. Both parties must follow the instructions in Join the Participant or Scaled-Out OSNs to the Founder's Ordering Service all the way to completion or the founder won't be able to manage the network.

Join the Participant or Scaled-Out OSNs to the Founder's Ordering Service

When you provision a participant instance, it is created with 3 orderers. The orderers are inactive until they are joined to a network. When you scale out a founder, the new orderers are also inactive until they are joined to a network.

To add or remove multiple orderers, add or remove one at a time and ensure that the network has returned to operational status before making the next change. See What is the Ordering Service? for additional important details about adding, removing, starting, and stopping Raft orderers.

Export the OSN Settings From the Participant or Scaled-Out Orderers

To join the participant or scaled-out orderers to a network, export their settings and import them into the founder.

1. In the participant console (or the founder console for scaled-out orderers), on the Node tab find the orderer node (or the first orderer node if multiple nodes exist). Select the Action menu for this node and select **Export OSN Settings**.

   This generates a JSON file with the settings and saves the file. The file contains the organization's certificate and the selected orderer service node (OSN) settings signed by the private key of the administrator of the participant organization. Send this file to the administrator of the founder instance.

   Applications that are run on channels that use this OSN also require this exported TLS certificate.

2. In the founder console, open the Network tab. Click **Add OSN**. A window opens prompting you for the location of the JSON file that was provided by the participant. Select to upload the file and click **Add**.

   The participant organization or newly scaled-out orderer is added to the orderer organization section of the system channel list.

Export the Founder's Configuration Settings

After the participant or scaled-out orderers have been added to the founder, you must export the founder's settings and import them into the participant or scaled-out orderer.

1. In the founder console, open the Network tab. Click **Export Network Config Block**.

   The network configuration block contains the latest system channel configuration block. This can be saved and sent to the participant administrator.

2. In the participant console (or the founder console for scaled-out orderers), on the Node tab find the orderer node (or the first orderer node if multiple nodes exist). Select the Action menu for this node and select **Import Network Config Block**.

   You are prompted for the file that was sent by the founder instance administrator.

3. In the participant console, refresh the Node tab. The orderer node status will be listed as down. From the Action menu select **Start**.

   Each orderer node that is started will be added to the Raft cluster in the founder.

Each time a new OSN is added by scaling out the orderer (as described in Scale Your Instance) these steps must be repeated to add the new OSN to the Raft cluster.

> **Note**
>
> You can't add multiple OSNs to a network in a single batch. Ensure that only one OSN is added at a time.

Edit Ordering Service Settings for the Network

You can update the ordering service settings for the founder instance.

- The updated settings are used when you create channels and are not applied to existing channels.
- You can update the ordering service settings for an individual existing channels as described in Edit Ordering Service Settings for a Channel.
- If you change the ordering service settings and there are applications running against the network, those applications must be manually updated to use the revised ordering service settings.
- It isn't common, but in some situations, you might expose a different ordering service to some of the network participants. In this case, you'll export the updated network config block and the required participants will import the revised settings. See Join the Participant or Scaled-Out OSNs to the Founder's Ordering Service.

You must be an administrator to complete this task.

1. Go to the founder's console and click the **Network** tab.
2. Click the **Ordering Service Settings** button.

   The Ordering Service Settings dialog box is displayed.
3. Update the settings as needed.

Field	Description
Batch Timeout (ms)	Specify the amount of time in milliseconds that the system will wait before creating a batch. Enter a number between 1 and 3600000.
Max Message Count	Specify the maximum number of message to include in a batch. Enter a number between 1 and 4294967295.
Absolute Message Bytes	Specify the maximum number of bytes allowed for the serialized messages in a batch. This number must be larger than the value you enter in the Preferred Message Bytes field.

Field	Description
Preferred Message Bytes	Specify the preferred number of bytes allowed for the serialized messages in a batch. A message larger than this size results in a larger batch, but the batch size will be equal to or less than the number of bytes you specified in the Absolute Message Bytes field. Typically you set this value to 1 MB or less. The value that you enter in this field must be smaller than the value you enter in the Absolute Message Bytes field.
Snapshot Interval Size	Defines number of MB per which a snapshot is taken.

4. Click **Update**.

   The updated settings are saved.

View Ordering Service Settings

You can view the founder's ordering service settings that were imported into a participant's Oracle Blockchain Platform instance.

If the founder changes the ordering service settings the new settings must be ported to the participant as described in Join the Participant or Scaled-Out OSNs to the Founder's Ordering Service. If there are applications running against the network, then those applications must be manually updated to use the revised ordering service settings.

1. Go to the participant's console and select the Network tab.
2. Click **Ordering Service Settings** and click **View**.

   The Ordering Settings dialog is displayed.

4

Understand and Manage Nodes by Type

This topic contains information to help you understand the different node types and where you can get more information about how the nodes are performing in the network.

Topics:

Manage CA Nodes

This topic contains information about certificate authority (CA) nodes, including how to view and edit the CA node configuration.

View and Edit the CA Node Configuration

A certificate authority (CA) node's configuration determines how the node performs and behaves on the network.

Only administrators can change a node's configuration. If you've got user permissions, then you can view a node's configuration settings. See CA Node Attributes.

1. Go to the console and select the Nodes tab.
2. In the Nodes tab, go to the Nodes table, locate the CA node that you want configuration information for, and click the node's **More Actions** button.
3. The configuration option is determined by your permissions. If you're an administrator, locate and click **Edit Configuration**. If you're a user, locate and click **View**.

   The Configure dialog is displayed.
4. If you're an administrator, then modify the node's settings as needed.
5. Click **Submit** to save the configuration changes, or click **X** to close the Configure dialog.
6. Restart the node to apply any changes that you made.

Manage the Console Node

This topic contains information about the console node, including how to view and edit the console node configuration.

View and Edit the Console Node Configuration

The console node's configuration determines how it performs and behaves on the network.

Only administrators can change a node's configuration. If you've got user permissions, then you can view a node's configuration settings. See Console Node Attributes.

1. Go to the console and select the Nodes tab.
2. In the Nodes tab, go to the Nodes table, locate the console node and click its **More Actions** button.
3. The configuration option is determined by your permissions. If you're an administrator, locate and click **Edit Configuration**. If you're a user, locate and click **View**.

   The Configure dialog is displayed.
4. If you're an administrator, then modify the node's settings as needed.
5. Click **Submit** to save the configuration changes, or click **X** to close the Configure dialog.
6. Restart the node to apply any changes that you made.

Manage Orderer Nodes

This topic contains information about ordering service nodes (OSNs), including how to view and edit OSNs and how to add an additional OSN.

View and Edit the Orderer Node Configuration

An orderer node's configuration determines how the node performs and behaves on the network.

Only administrators can change a node's configuration. If you've got user permissions, then you can view a node's configuration settings. See Orderer Node Attributes.

1. Go to the console and select the Nodes tab.
2. In the Nodes tab, go to the Nodes table, locate the orderer node that you want configuration information for, and click the node's **More Actions** button.
3. The configuration option is determined by your permissions. If you're an administrator, locate and click **Edit Configuration**. If you're a user, locate and click **View**.

   The Configure dialog is displayed.
4. If you're an administrator, then modify the node's settings as needed.
5. Click **Submit** to save the configuration changes, or click **X** to close the Configure dialog.
6. Restart the node to apply any changes that you made.

Add an Orderer Node

Founder instances are provisioned with 3 OSNs, all of which are active after instance creation. Additional OSNs can be scaled out as described in Scale Your Instance. These OSNs will not be started automatically. You must start them and export the updated network configuration block to the partipant instances as described in Join the Participant or Scaled-Out OSNs to the Founder's Ordering Service.

Participant instances are created with 3 OSNs, but none of these OSNs are joined to the network or started when the instance is provisioned. You must follow the instructions in Join the Participant or Scaled-Out OSNs to the Founder's Ordering Service in order to join them to the network and start the nodes. If you want to scale out the participant OSNs these steps must be repeated.

Manage Peer Nodes

This topic contains information about peer nodes, including how to view and edit peer nodes and how to get a list of chaincodes that are installed on a peer.

View and Edit the Peer Node Configuration

A peer node's configuration determines how the node performs and behaves on the network.

Only administrators can change a node's configuration. If you've got user permissions, then you can view a node's configuration settings. See Peer Node Attributes.

1. Go to the console and select the Nodes tab.
2. In the Nodes tab, go to the Nodes table, locate the peer node that you want configuration information for, and click the node's **More Actions** button.
3. The configuration option is determined by your permissions. If you're an administrator, locate and click **Edit Configuration**. If you're a user, locate and click **View**.

   The Configure dialog is displayed.
4. If you're an administrator, then modify the node's settings as needed.
5. Click **Submit** to save the configuration changes, or click **X** to close the Configure dialog.
6. Restart the node to apply any changes that you made.

List Chaincodes Installed on a Peer Node

You can view a list of the chaincodes and their versions installed on a specific peer node in your network.

If you don't see the chaincode or the chaincode version you were expecting, then you can install a chaincode or upgrade a chaincode to the peer node. You must be an administrator to install or upgrade a chaincode.

1. Go to the console and select the Nodes tab.
2. In the Nodes tab, click the name of the peer node you want to see information for.

   The Node Information page is displayed.
3. Click the **Chaincodes** pane to view a list of chaincodes installed on the selected peer node.

Manage REST Proxy Nodes

This topic contains information to help you understand how the REST proxy is used, add enrollments to the REST proxy, and view and edit the REST proxy nodes.

How's the REST Proxy Used?

The REST proxy maps an application identity to a blockchain member, which enables users and applications to call the Oracle Blockchain Platform REST APIs.

Instead of using the native Hyperledger Fabric APIs, Oracle Blockchain Platform can use the REST proxy to interact with the Hyperledger Fabric network. When you use the native Hyperledger Fabric APIs, you connect to the peers and orderer directly. However, the REST proxy lets you query or call a Hyperledger Fabric chaincode via the RESTful protocol.

Add Enrollments to the REST Proxy

You can add Hyperledger Fabric enrollments to the REST proxy. Enrollments allow users to call the REST proxy without an enrollment certificate.

If you want to add a user to an enrollment, they must already exist in IDCS, and be assigned to the REST_USER role.

Use the Blockchain Platform console to add new enrollments and associate IDCS users with these enrollments. The enrollments are managed entirely within Blockchain Platform, not within IDCS.

For information about how users access the REST resources, see REST API for Oracle Blockchain Platform.

1. Go to the console and select the Nodes tab.
2. In the Nodes tab, find the REST proxy node you want to add an enrollment to, and click the **Action** menu for this node.
3. Click **View or Manage Enrollments** to see a list of the node's current enrollments.

   A list of the current enrollments is displayed. You can delete existing enrollments as well as adding new ones from this page.
4. Expand **Create New Enrollment**.
5. In the **Enrollment ID** field, enter the name of the enrollment to add.

   The enrollment ID can include only alphanumeric characters, hyphens (-), and underscores (_).
6. Optionally, in the **User ID** field, enter the ID of a user with the REST_USER role to associate with the enrollment. Click **Enroll**.

   After you click **Enroll**:

   - The enrollment is created and displays in the Enrollments table.
   - The new enrollment is copied to each REST Proxy node in the network.
   - If you specified a user ID, that ID is associated with the enrollment, and cannot be removed from the associated REST users list. If the user ID is not a valid REST user, an error is returned.
   - If you specified a user ID, the generated enrollment certificate includes the ID as the `username` attribute.
   - User IDs that contain a colon (:) are not supported for REST API calls that use basic authentication. You can use basic authentication for testing and internal development purposes. Do not use basic authentication in production environments.

7. In the Associated REST Client Users pane you can view and manage any users associated with a current enrollment, including deleting a user from an enrollment.
8. Add another user to the enrollment by expanding **Associate New Users**. Enter the email or ID of a user that is already assigned the REST_USER role. Click **Associate**.

After you've created an enrollment and associated a user with it, when you use REST to run transactions on the blockchain the initiator listed in the details of the block will be listed as the new enrollment rather that the original default user.

View and Edit the REST Proxy Node Configuration

A REST proxy node's configuration determines how the node performs and behaves on the network.

Only administrators can change a node's configuration. If you've got user permissions, then you can view a node's configuration settings. See REST Proxy Node Attributes.

1. Go to the console and select the Nodes tab.
2. In the Nodes tab, go to the Nodes table, locate the REST proxy node that you want configuration information for, and click the node's **More Actions** button.
3. The configuration option is determined by your permissions. If you're an administrator, locate and click **Edit Configuration**. If you're a user, locate and click **View**.

   The Configure dialog is displayed.
4. If you're an administrator, then modify the node's **Proposal Wait Time (ms)**, **Transaction Wait Time (ms)**, **Log Level**, and **Transaction Event Logging** attributes as needed.
5. Click **Submit** to save the configuration changes, or click **X** to close the Configure dialog.

5

Extend the Network

This topic contains information to help founders add organizations to the blockchain network. This topic also contains information to help organizations join a network.

Topics

Add Oracle Blockchain Platform Participant Organizations to the Network

This topic contains information about joining an Oracle Blockchain Platform participant organization to an Oracle Blockchain Platform network.

Typical Workflow to Join a Participant Organization to an Oracle Blockchain Platform Network

Here are the tasks the founder and participants organizations need to perform to set up a blockchain network.

Adding Participant Organizations to a Blockchain Network

Task	Who Does This?	Description	More Information
Export the participant organization's certificates and import them into the network	Participant organization outputs certificates Founder organization uploads certificates	In the participant organization's instance, use the wizard to output the certificates into a JSON file and send them to the founder organization. The founder uploads the certificates to add the participant to the network.	Import Certificates to Add Organizations to the Network
Export the participant organization's ordering service node (OSN) settings and send to the founder administrator	Participant organization outputs a settings file Founder organization uploads the settings	In the participant organization's instance, export the settings into a JSON file and sends them to the founder organization. The founder uploads the settings to add the ordering service.	Join the Participant or Scaled-Out OSNs to the Founder's Ordering Service

Task	Who Does This?	Description	More Information
Export the founder organization's network configuration block and upload it to the participant organization	Founder organization exports network configuration block information Participant organization uploads network configuration block information	In the founder's instance, download the network configuration block information (JSON file). Then in the participant's instance, upload the network configuration block.	Join the Participant or Scaled-Out OSNs to the Founder's Ordering Service

Join Participant Organizations to the Channel and Set Anchor Peers

Task	Who Does This?	Description	More Information
Create a channel	Founder organization	In the founder's instance, create a channel that the founder and participants use to communicate. Add the founder's peers to the channel. You must select any newly added participants and assign them permissions on the channel. Note that instead of creating a new channel, you can add participants to an existing channel.	Create a Channel
Join participants to the channel	Participant organization	In the participant's instance, join the channel that was created in the founder's instance.	Join a Peer to a Channel
Set anchor peers on the founder and participants	Founder organization Participant organization	In the founder and participant instances, specify which peers you want to use as anchor peers. You must select at least one anchor peer for each member.	Add an Anchor Peer

Deploy the Chaincode Across the Blockchain Network

Task	Who Does This?	Description	More Information
Install the chaincode on the founder	Founder organization	In the founder's instance, upload and install the chaincode. Choose the peers to install the chaincode on.	Use Quick Deployment
Deploy the chaincode and specify an endorsement policy on the founder	Founder organization	In the founder's instance, deploy the chaincode to activate it on the network. An endorsement policy is required to specify the number of members that must approve chaincode transactions before they're submitted to the ledger.	• Deploy a Chaincode • Specify an Endorsement Policy

Task	Who Does This?	Description	More Information
Install the chaincode on the participant	Participant organization	In the participant's instance, install the chaincode that your network will use. Because you'll install the same chaincode that you installed and deployed on the founder, you don't need to deploy the chaincode on the participant. When the participant installs the chaincode, it's already deployed.	Use Quick Deployment

Run Transactions

Task	Who Does This?	Description	More Information
Invoke the chaincode and monitor network activity and ledger updates	Founder organization Participant organization	Begin using your network's chaincode for transactions. Both the founder and the participants can use their consoles to find out information about the activity on the network. Specifically, you can use the console's Channels tab to locate information about specific ledger transactions	• Find Information About Nodes • View a Channel's Ledger Activity

Add Fabric Organizations to the Network

This topic contains information about joining Hyperledger Fabric organizations to an Oracle Blockchain Platform network.

Typical Workflow to Join a Fabric Organization to an Oracle Blockchain Platform Network

A Fabric organization and the Oracle Blockchain Platform founder organization must complete the following tasks to join the Fabric organization to the Oracle Blockchain Platform network.

Task	Who Does This?	Description	More Information
Create the certificate file for the Fabric organization	Fabric organization	Find the Fabric organization's Admin, CA, and TLS certificate information and use it to compose a JSON certificates file.	Create a Fabric Organization's Certificates File
Upload Fabric organization's certificate file to the Oracle Blockchain Platform network	Founder organization	Use the console to upload and import the Fabric organization's certificate file to add the Fabric organization to the network.	Import Certificates to Add Organizations to the Network

Task	Who Does This?	Description	More Information
Create a channel	Founder organization	Create a channel and add the Fabric organization to it.	Create a Channel
Export the ordering service settings from founder	Founder organization	Wrote the founder's ordering services settings to a JSON file and send the file to the Fabric organization.	Join the Participant or Scaled-Out OSNs to the Founder's Ordering Service

Compose an orderer certificate file	Fabric organization	Create a file named `orderer.pem` that includes the `tlscacert` information. Go to the exported ordering service settings file and copy the `tlscacert` information. After you paste the `tlscacert` information into the `orderer.pem` file, you must replace all instances of \n with the newline character.
		The `orderer.pem` file must have the following format:
		`-----BEGIN CERTIFICATE-----`
		`...`
		`...`
		`...`
		`-----END CERTIFICATE-----`
			Create a Fabric Organization's Certificates File

Provide ordering service settings	Founder organization	Open the ordering service settings file and find the ordering service's address and port and give them to the Fabric organization, as shown in the following example:
		`"orderingServiceNodes": [`
		`{`
		`"address": "grpcs://example_address:7443",`
		`...`
		`}]`
			NA

Add the Fabric organization to the network	Fabric organization	The Fabric organization copies certificates into its environment, sets environment variables, fetches the genesis block, joins the channel, and installs the chaincode.	Prepare the Fabric Environment to Use the Oracle Blockchain Platform Network

Create a Fabric Organization's Certificates File

For a Fabric organization to join an Oracle Blockchain Platform network, it must write a certificates file containing its admincerts, cacerts, and tlscacerts information. The Oracle Blockchain Platform founder organization imports this file to add the Fabric organization to the network.

The Fabric certificates information is stored in PEM files located in the Fabric organization's MSP folder. For example, `network_name_example/crypto-config/peerOrganizations/example_org.com/msp/`.

The certificates file must be in written in JSON and must contain the following fields. For all certificates, when you copy the certificate information into the JSON file, you must replace each new line with `\n`, so that the information is all on one line with no spaces, as shown in the following example.

- **mspid** — Specifies the name of the Fabric organization.
- **type** — Indicates that the organization is a network participant. This value must be `Participant`.
- **admincert** — Contains the contents of the organization's Admin certificates file: `Admin@example_org.com-cert.pem`.
- **cacert** — Contains the contents of the organization's CA certificates file: `ca.example_org-cert.pem`.
- **tlscacert** — Contains the contents of the organization's TLS certificate file: `tlsca.example_org-cert.pem`.
- **intermediatecerts**— This optional element contains the contents of an intermediate CA certificates file. Do not specify this element unless there is an intermediate CA certificates file.
- **nodeouidentifiercert**— This section contains certificates that identify Node OU roles.
- **adminouidentifiercert**— Contains the contents of the organization's certificate file that is used to identify Node OU admin roles. If you do not need the admin role, you can use the cacert file contents, or intermediate certificate file contents, as the adminouidentifier contents.
- **clientouidentifiercert**— Contains the contents of the organization's certificate file that is used to identify Node OU client roles.
- **ordererouidentifiercert**— Contains the contents of the organization's certificate file that is used to identify Node OU orderer roles. If you do not need the orderer role, you can use the cacert file contents, or intermediate certificate file contents, as the ordererouidentifier contents.
- **peerouidentifiercert**— Contains the contents of the organization's certificate file used to identify Node OU peer roles.

Structure the file similar to the following example:

```
{
  "mspID": "examplemspID",
  "type":  "Participant",
  "certs": {
   "admincert": "-----BEGIN CERTIFICATE-----
\nexample_certificate\nexample_certificate==\n-----END CERTIFICATE-----\n",
   "cacert": "-----BEGIN CERTIFICATE-----
```

```
\nexample_certificate\nexample_certificate==\n-----END CERTIFICATE-----\n",
   "tlscacert": "-----BEGIN CERTIFICATE-----
\nexample_certificate\nexample_certificate==\n-----END CERTIFICATE-----\n"
   "nodeouidentifiercert": {
      "adminouidentifiercert": "-----BEGIN CERTIFICATE-----
\nexample_certificate\nexample_certificate==\n-----END CERTIFICATE-----\n",
      "clientouidentifiercert": "-----BEGIN CERTIFICATE-----
\nexample_certificate\nexample_certificate==\n-----END CERTIFICATE-----\n",
      "ordererouidentifiercert": "-----BEGIN CERTIFICATE-----
\nexample_certificate\nexample_certificate==\n-----END CERTIFICATE-----\n",
      "peerouidentifiercert": "-----BEGIN CERTIFICATE-----
\nexample_certificate\nexample_certificate==\n-----END CERTIFICATE-----\n"
    }
  }
}
```

Prepare the Fabric Environment to Use the Oracle Blockchain Platform Network

You must modify the Fabric organization's environment before it can use the Oracle Blockchain Platform network.

Confirm that the following prerequisite tasks were completed. For more information, see Typical Workflow to Join a Fabric Organization to an Oracle Blockchain Platform Network.

- The Fabric organization's certificate file was created and sent to the Oracle Blockchain Platform network founder.
- The network founder uploaded the certificates file to add the Fabric organization to the network.
- The network founder created a new channel and added the Fabric organization to it.
- The network founder downloaded its ordering service settings and sent them to the Fabric organization.
- The Fabric organization created the orderer certificate file.
- The network founder gave the ordering service address and port to the Fabric organization.

You must add the Fabric organization and install and test the chaincode.

1. Navigate to the Fabric network directory and launch the peer container.
2. Fetch the channel's genesis block with this command:

   ```
   peer channel fetch 0 mychannel.block -o ${orderer_addr}:${orderer_port} -c
   mychannel --tls --cafile orderer.pem --logging-level debug
   ```

   Where:

   - `{orderer_addr}` is the Founder's orderer address.
   - `{orderer_port}` is the Founder's port number.
   - `-c mychannel` is the name of the channel that the Founder created. This is the channel where the Fabric organization will send and receive transactions on the Oracle Blockchain Platform network.

- `orderer.pem` is the Founder's orderer certificate file.

3. Join the channel with this command:

```
peer channel join -b mychannel.block -o ${orderer_addr}:${orderer_port} --
tls --cafile orderer.pem --logging-level debug
```

4. Install the chaincode with this command:

```
peer chaincode install -n mycc -v 1.0 -l "golang" -p ${CC_SRC_PATH}
```

   Where `CC_SRC_PATH` is the folder that contains the chaincode.

5. Instantiate the chaincode with this command:

```
peer chaincode instantiate -o  ${orderer_addr}:${orderer_port} --tls --
cafile orderer.pem -C mychannel -n mycc -l golang -v 1.0 -c '{"Args":
["init","a","100","b","200"]}' -P <policy_string> --logging-level debug
```

6. Invoke the chaincode with this command:

```
peer chaincode invoke -o ${orderer_addr}:${orderer_port}  --tls true --
cafile orderer.pem -C mychannel -n mycc -c '{"Args":
["invoke","a","b","10"]}' --logging-level debug
```

7. Query the chaincode with this command:

```
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'  --
logging-level debug
```

Add Organizations with Third-Party Certificates to the Network

This topic contains information about joining organizations using third-party certificates to an Oracle Blockchain Platform network.

Typical Workflow to Join an Organization With Third-Party Certificates to an Oracle Blockchain Platform Network

Organizations with certificates issued by a third-party certificate authority (CA) can join the Oracle Blockchain Platform network as participants.

Client-only Organizations

These participants are client-only organizations and have no peers or orderers. They cannot create channels, join peers or install chaincode.

After joining the network, these organizations can use an SDK or a Hyperledger Fabric CLI to complete the following tasks:

- Deploy, invoke, and query chaincode if they're a client organization administrator.
- Invoke and query chaincode if they're a client organization non-administrator.

The following users can take steps to control who can deploy and invoke chaincode when client-only organizations are part of the network:

- The chaincode owner who installs the chaincode onto peers can decide who can deploy the chaincode by using the Hyperledger Fabric `peer chaincode package -i` instantiation policy command to set the instantiation policy for the chaincode.
- The chaincode instantiator can use the Hyperledger Fabric `peer chaincode instantiate -P` endorsement policy command to set the endorsement policy controlling who can invoke the chaincode.
- The channel owner can decide who can call or query a chaincode by setting the channel proposal and query access control list. For more information, see Hyperledger Fabric Access Control Lists.

Workflow

The organization with third-party certificates and the Oracle Blockchain Platform founder must complete the following tasks to join the organization to an Oracle Blockchain Platform network.

Task	Who Does This?	Description	More Information
Get the third-party certificates	Third-party certificates (participant) organization	Go to the third-party CA server and generate the required certificates files. Format the files as needed to import them into the network.	Third-Party Certificate Requirements
Create the certificates file for import	Third-party certificates (participant) organization	Find the participant's Administrator and CA certificate information and use it to compose a JSON certificates file.	Create an Organization's Third-Party Certificates File
Upload a certificate file for the third-party (participant) organization	Founder organization	Use the console to upload and import the participant's certificate file to add the participant to the network.	Import Certificates to Add Organizations to the Network
Export the ordering service settings from network founder and provide them to the third-party (participant) organization	Founder organization	Write the founder's ordering services settings to a JSON file and send the file to the participant. Open the ordering service settings file and find the ordering service's address and port and give them to the participant, as shown in the following example: `"orderingServiceNodes ": [ { "address": "grpcs:// example_address:7443" ... }]`	Join the Participant or Scaled-Out OSNs to the Founder's Ordering Service
Create the channel	Founder organization	Create a channel and add the participant to it.	Create a Channel

Task	Who Does This?	Description	More Information
Install and deploy the chaincode	Founder organization	In the founder's instance, upload, install, and deploy the chaincode. Choose the network peers to install the chaincode on.	Use Quick Deployment
Set up the third-party (participant) organization's environment	Third-party certificates (participant) organization	To query or call chaincodes, the participant must complete the following steps: • Add the founder's ordering service's address and port to the participant's environment. • Configure the environment to use Hyperledger Fabric CLI or SDKs. • Install the chaincode on peers.	Prepare the Third-Party Environment to Use the Oracle Blockchain Platform Network

Third-Party Certificate Requirements

To successfully join the network, an organization must generate the required third-party certificates. The information in these certificates is used to create the organization's certificates file, which is then imported into the founder's instance.

Which Certificates Do Organizations Need to Provide?

You must generate the following certificates from your certificate authority (CA) server:

- Client Public Certificate
- CA Root Certificate

What Are the Requirements for These Certificates?

The certificates must meet the following requirements:

- When generating the private key, you must use the Elliptic Curve Digital Signature Algorithm (ECDSA). ECDSA is the only accepted algorithm for Hyperledger Fabric membership service provider (MSP) keys.
- The Subject Key Identifier (SKI) is mandatory and you must indicate it as x509 extensions in the extension file.
- You must convert the key files from the .key to the .pem format.
- You must convert the certificates from the .crt to the .pem format.

Creating the Certificates

The following examples show how to use OpenSSL or the Hyperledger Fabric `cryptogen` utility to generate your certificates. For more information on the commands used, see the following websites:

- OpenSSL documentation
- cryptogen utility documentation

Complete the following steps to create your certificates using OpenSSL.

1. Run the following commands to create a self-signed CA certificate and key:

```
openssl ecparam -name prime256v1 -genkey -out ca.key
openssl pkcs8 -topk8 -inform PEM -in ca.key -outform pem -nocrypt -out ca-
key.pem
openssl req -new -key ca-key.pem -out ca.csr
openssl x509 -req -days 365 -in ca.csr -signkey ca-key.pem -out ca.crt -
extensions x509_ext -extfile opensslca.conf
openssl x509 -in ca.crt -out ca.pem -outform PEM
```

The following text shows an example `opensslca.conf` file.

```
[ req ]
default_bits        = 2048
distinguished_name  = subject
req_extensions      = req_ext
x509_extensions     = x509_ext
string_mask         = utf8only


[ subject ]
countryName         = CN
#countryName_default     = US


stateOrProvinceName     = Beijing
#stateOrProvinceName_default = NY

localityName            = Beijing
#localityName_default        = New York

organizationName         = thirdpartyca, LLC
#organizationName_default    = Example, LLC

# Use a friendly name here because its presented to the user. The server's
DNS
#   names are placed in Subject Alternate Names. Plus, DNS names here is
deprecated
#   by both IETF and CA/Browser Forums. If you place a DNS name here, then
you
#   must include the DNS name in the SAN too (otherwise, Chrome and others
that
#   strictly follow the CA/Browser Baseline Requirements will fail).
commonName          = thirdpartyca
#commonName_default      = Example Company

emailAddress            = ca@thirdpartyca.com



# Section x509_ext is used when generating a self-signed certificate.
I.e., openssl req -x509 ...
[ x509_ext ]
```

```
subjectKeyIdentifier        = hash
authorityKeyIdentifier  = keyid,issuer

# You only need digitalSignature below. *If* you don't allow
#   RSA Key transport (i.e., you use ephemeral cipher suites), then
#   omit keyEncipherment because that's key transport.
basicConstraints        = CA:TRUE
keyUsage            = Certificate Sign, CRL Sign, digitalSignature,
keyEncipherment
subjectAltName          = @alternate_names
nsComment           = "OpenSSL Generated Certificate"

# RFC 5280, Section 4.2.1.12 makes EKU optional
#   CA/Browser Baseline Requirements, Appendix (B)(3)(G) makes me confused
#   In either case, you probably only need serverAuth.
# extendedKeyUsage  = serverAuth, clientAuth

# Section req_ext is used when generating a certificate signing request.
I.e., openssl req ...
[ req_ext ]

subjectKeyIdentifier        = hash

basicConstraints        = CA:FALSE
keyUsage            = digitalSignature, keyEncipherment
subjectAltName          = @alternate_names
nsComment           = "OpenSSL Generated Certificate"

# RFC 5280, Section 4.2.1.12 makes EKU optional
#   CA/Browser Baseline Requirements, Appendix (B)(3)(G) makes me confused
#   In either case, you probably only need serverAuth.
# extendedKeyUsage  = serverAuth, clientAuth

[ alternate_names ]

DNS.1       = localhost
DNS.2       = thirdpartyca.com
#DNS.3       = mail.example.com
#DNS.4       = ftp.example.com

# Add these if you need them. But usually you don't want them or
#   need them in production. You may need them for development.
# DNS.5       = localhost
# DNS.6       = localhost.localdomain
# DNS.7       = 127.0.0.1
```

2. Run the following commands to create a user certificate and key using the previous CA key.

```
openssl ecparam -name prime256v1 -genkey -out user.key
openssl pkcs8 -topk8 -inform PEM -in user.key -outform pem -nocrypt -out
user-key.pem
openssl req -new -key user-key.pem -out user.csr
openssl x509 -req -days 365 -sha256 -CA ca.pem -CAkey ca-key.pem -CAserial
ca.srl -CAcreateserial -in user.csr -out user.crt -extensions x509_ext -
```

```
extfile openssl.conf
openssl x509 -in user.crt -out user.pem -outform PEM
```

The following text shows an example `openssl.conf` file.

```
[ req ]
default_bits        = 2048
default_keyfile     = tls-key.pem
distinguished_name  = subject
req_extensions      = req_ext
x509_extensions     = x509_ext
string_mask         = utf8only

# The Subject DN can be formed using X501 or RFC 4514 (see RFC 4519 for a
description).
# Its sort of a mashup. For example, RFC 4514 does not provide
emailAddress.
[ subject ]
countryName         = CN
#countryName_default     = US

stateOrProvinceName     = Beijing
#stateOrProvinceName_default = NY

localityName            = Beijing
#localityName_default        = New York

organizationName         = thirdpartyca, LLC
#organizationName_default    = Example, LLC

# Use a friendly name here because its presented to the user. The server's
DNS
#   names are placed in Subject Alternate Names. Plus, DNS names here is
deprecated
#   by both IETF and CA/Browser Forums. If you place a DNS name here, then
you
#   must include the DNS name in the SAN too (otherwise, Chrome and others
that
#   strictly follow the CA/Browser Baseline Requirements will fail).
commonName          = admin@thirdpartyca.com
#commonName_default      = Example Company

emailAddress            = admin@thirdpartyca.com
#emailAddress_default        = test@example.com

# Section x509_ext is used when generating a self-signed certificate.
I.e., openssl req -x509 ...
[ x509_ext ]
subjectKeyIdentifier        = hash
authorityKeyIdentifier  = keyid,issuer

# You only need digitalSignature below. *If* you don't allow
#   RSA Key transport (i.e., you use ephemeral cipher suites), then
#   omit keyEncipherment because that's key transport.
basicConstraints        = CA:FALSE
```

```
keyUsage            = digitalSignature, keyEncipherment

subjectAltName          = @alternate_names
nsComment           = "OpenSSL Generated Certificate"

# RFC 5280, Section 4.2.1.12 makes EKU optional
#   CA/Browser Baseline Requirements, Appendix (B)(3)(G) makes me confused
#   In either case, you probably only need serverAuth.
#extendedKeyUsage  = Any Extended Key Usage
#extendedKeyUsage = serverAuth, clientAuth


# Section req_ext is used when generating a certificate signing request.
I.e., openssl req ...
[ x509_ca_ext ]
subjectKeyIdentifier        = hash
authorityKeyIdentifier  = keyid,issuer


# You only need digitalSignature below. *If* you don't allow
#   RSA Key transport (i.e., you use ephemeral cipher suites), then
#   omit keyEncipherment because that's key transport.
basicConstraints        = CA:TRUE
keyUsage            = Certificate Sign, CRL Sign, digitalSignature,
keyEncipherment
subjectAltName          = @alternate_names
nsComment           = "OpenSSL Generated Certificate"


# RFC 5280, Section 4.2.1.12 makes EKU optional
#   CA/Browser Baseline Requirements, Appendix (B)(3)(G) makes me confused
#   In either case, you probably only need serverAuth.
#extendedKeyUsage  = Any Extended Key Usage
extendedKeyUsage = serverAuth, clientAuth


# Section req_ext is used when generating a certificate signing request.
I.e., openssl req ...
[ req_ext ]
subjectKeyIdentifier        = hash
basicConstraints        = CA:FALSE
keyUsage            = digitalSignature, keyEncipherment
subjectAltName          = @alternate_names
nsComment           = "OpenSSL Generated Certificate"


# RFC 5280, Section 4.2.1.12 makes EKU optional
#   CA/Browser Baseline Requirements, Appendix (B)(3)(G) makes me confused
#   In either case, you probably only need serverAuth.
#extendedKeyUsage  = Any Extended Key Usage
#extendedKeyUsage = serverAuth, clientAuth

[ alternate_names ]
DNS.1       = localhost
DNS.3       = 127.0.0.1
DNS.4       = 0.0.0.0
```

```
   # Add these if you need them. But usually you don't want them or
   #   need them in production. You may need them for development.
   # DNS.5       = localhost
   # DNS.6       = localhost.localdomain
   # DNS.7       = 127.0.0.1
   # IPv6 localhost
   # DNS.8     = ::1
```

To create your certificates using the Hyperledger Fabric `cryptogen` utility, run the following command.

```
cryptogen generate --config=./crypto-config.yaml
```

The following text shows an example `crypto-config.yaml` file.

```
# Copyright IBM Corp. All Rights Reserved.
#
# SPDX-License-Identifier: Apache-2.0
#

# ---------------------------------------------------------------------------
# "PeerOrgs" - Definition of organizations managing peer nodes
# ---------------------------------------------------------------------------
PeerOrgs:
  #
---------------------------------------------------------------------------
  # Org1
  #
---------------------------------------------------------------------------
  - Name: Org1
    Domain: org1.example.com
    EnableNodeOUs: true
    #
---------------------------------------------------------------------------
    # "Specs"
    #
---------------------------------------------------------------------------
    # Uncomment this section to enable the explicit definition of hosts in
your
    # configuration.  Most users will want to use Template, below
    #
    # Specs is an array of Spec entries.  Each Spec entry consists of two
fields:
    #   - Hostname:   (Required) The desired hostname, sans the domain.
    #   - CommonName: (Optional) Specifies the template or explicit override
for
    #                 the CN.  By default, this is the template:
    #
    #                              "{{.Hostname}}.{{.Domain}}"
    #
    #                 which obtains its values from the Spec.Hostname and
    #                 Org.Domain, respectively.
    #
---------------------------------------------------------------------------
```

```
    # Specs:
    #   - Hostname: foo # implicitly "foo.org1.example.com"
    #     CommonName: foo27.org5.example.com # overrides Hostname-based FQDN
set above
    #   - Hostname: bar
    #   - Hostname: baz
    #
--------------------------------------------------------------------------------
    # "Template"
    #
--------------------------------------------------------------------------------
    # Allows for the definition of 1 or more hosts that are created
sequentially
    # from a template. By default, this looks like "peer%d" from 0 to Count-1.
    # You may override the number of nodes (Count), the starting index (Start)
    # or the template used to construct the name (Hostname).
    #
    # Note: Template and Specs are not mutually exclusive.  You may define
both
    # sections and the aggregate nodes will be created for you.  Take care
with
    # name collisions
    #
--------------------------------------------------------------------------------
    Template:
      Count: 2
      # Start: 5
      # Hostname: {{.Prefix}}{{.Index}} # default
    #
--------------------------------------------------------------------------------
    # "Users"
    #
--------------------------------------------------------------------------------
    # Count: The number of user accounts _in addition_ to Admin
    #
--------------------------------------------------------------------------------
    Users:
      Count: 1
  #
--------------------------------------------------------------------------------
  # Org2: See "Org1" for full specification
  #
--------------------------------------------------------------------------------
  - Name: Org2
    Domain: org2.example.com
    EnableNodeOUs: true
    Template:
      Count: 2
    Users:
      Count: 1
```

What's Next?

After confirming that you have generated and updated the proper files, you can then create the certificates file to import into the Oracle Blockchain Platform network. See Create an Organization's Third-Party Certificates File.

Create an Organization's Third-Party Certificates File

To join an Oracle Blockchain Platform network, the organization must write a certificates file containing its admincert and cacert information. The network founder imports this file to add the organization to the network.

Go to the certificates files that you generated from the CA server to find the information that you need to create the certificates file. See Third-Party Certificate Requirements.

The certificates file must be in written in JSON and contain the following fields:

- **mspid** — Specifies the name of the organization.
- **type** — Indicates that the organization is a network participant. This value must be `Participant`.
- **admincert** — Contains the contents of the organization's Admin certificates file. When you copy the certificates information into the JSON file, you must replace each new line with `\n`.
- **cacert** — Contains the contents of the organization's CA certificates file. When you copy the certificates information into the JSON file, you must replace each new line with `\n`.

This is how the file needs to be structured:

```
{
  "mspID": "examplemspID",
  "type":  "Participant",
  "certs": {
   "admincert": "-----BEGIN CERTIFICATE-----
\nexample_certificate\nexample_certificate==\n-----END CERTIFICATE-----\n",
   "cacert": "-----BEGIN CERTIFICATE-----
\nexample_certificate\nexample_certificate==\n-----END CERTIFICATE-----\n"
 }
}
```

Prepare the Third-Party Environment to Use the Oracle Blockchain Platform Network

You must set up the third-party organization's environment before it can use the Oracle Blockchain Platform network.

Confirm that the following prerequisite tasks were completed. For information, see Typical Workflow to Join an Organization With Third-Party Certificates to an Oracle Blockchain Platform Network.

- The third-party organization's certificate file was created and sent to the Oracle Blockchain Platform network founder.
- The network founder uploaded the certificates file to add the third-party organization to the network.
- The network founder exported the orderer service's settings and gave the service's address and port to the third-party organization and the organization added them to the environment.
- The network founder created a new channel and added the third-party organization to it.

- The network founder installed and instantiated the chaincode.

Setup organization's Environment

Before the third-party organization can successfully use the Oracle Blockchain Platform network, it must set up its environment to use Hyperledger Fabric CLI or SDKs. See the Hyperledger Fabric documentation.

Install the Chaincode

The third-party organization must install the chaincode on the peers. These peers must then be joined to the channel so that the chaincode can be invoked.

Deploy the Chaincode

If needed, the third-party organizations can deploy the chaincode on the channel. For example:

```
export  CORE_PEER_TLS_ENABLED=true
export  CORE_PEER_TLS_ROOTCERT_FILE=$PWD/tls-ca.pem
export  CORE_PEER_MSPCONFIGPATH=$PWD/crypto-config/peerOrganizations/
customerorg1.com/users/Admin@customerorg1.com/msp
export  CORE_PEER_LOCALMSPID="customerorg1"

### gets channel name from input###
CHANNEL_NAME=$1

echo "######### going to instantiate chaincode on channel ${CHANNEL_NAME}
##########"
CORE_PEER_ADDRESS=${peer_host}:${port} peer chaincode instantiate
-o ${peer_host}:${port}  --tls $CORE_PEER_TLS_ENABLED --cafile
./tls-ca.pem -C ${CHANNEL_NAME}  -n obcs-example02 -v v0 -c '{"Args":
["init","a","100","b","200"]}'
```

Invoke the Chaincode

Third-party organizations use the Hyperledger Fabric CLI or SDKs to invoke the chaincode. For example:

```
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_TLS_ROOTCERT_FILE=$PWD/tls-ca.pem
export CORE_PEER_MSPCONFIGPATH=$PWD/crypto-config/peerOrganizations/
customerorg1.com/users/User1@customerorg1.com/msp
export CORE_PEER_LOCALMSPID="customerorg1"

### gets channel name from input ###
CHANNEL_NAME=$1

#### do query or invoke on chaincode ####

CORE_PEER_ADDRESS=${peer_host}:${port} peer chaincode query -C
${CHANNEL_NAME} -n $2 -c '{"Args":["query","a"]}'

CORE_PEER_ADDRESS=${peer_host}:${port} peer chaincode invoke -o
${peer_host}:${port} --tls $CORE_PEER_TLS_ENABLED --cafile ./tls-
ca.pem -C ${CHANNEL_NAME} -n $2 -c '{"Args":["invoke","a","b","10"]}'
```

6

Develop Chaincodes

This topic contains information to help you understand how to write and test chaincodes for use in Oracle Blockchain Platform.

For information on developing chaincodes using low-code Blockchain App Builder, see Blockchain App Builder for Oracle Blockchain Platform.

Topics

Write a Chaincode

A chaincode is written in Go, Node.js, or Java and then packaged into a `.zip` file that is installed on the Oracle Blockchain Platform network.

Chaincodes define the data schema in the ledger, initialize it, complete updates when triggered by applications, and respond to queries. Chaincodes can also post events that allow applications to be notified and complete downstream operations. For example, after purchase orders, invoices, and delivery records have been matched by a chaincode, it can post an event so that a subscribing application can process related payments and update an internal ERP system.

Resources for Chaincode Development

Oracle Blockchain Platform uses Hyperledger Fabric as its foundation. Use the Hyperledger Fabric documentation to help you write valid chaincodes.

- Welcome to Hyperledger Fabric. Read the Key Concepts and Tutorials sections before you write you own chaincode.
- Go Programming Language. The Go compilers, tools, and libraries provide a variety of resources that simplify writing chaincodes.
- Package shim. The package shim provides APIs for the chaincode to access its state variables, get transaction context, and call other chaincodes. The package shim documentation describes the actual syntax that is required for your chaincode.

Oracle Blockchain Platform provides downloadable samples that help you understand how to write chaincodes and applications. See What Are Chaincode Samples?

You can add rich-query syntax to your chaincodes to query the state database. See SQL Rich Query Syntax and CouchDB Rich Query Syntax.

Package a Go Chaincode

After you've written your chaincode, compress it to a file in `.zip` format. You don't need to create a package for the Go chaincode or sign it — the Oracle Blockchain Platform installation and deployment process does this for you as described in Typical Workflow to Deploy Chaincodes.

If your chaincode has any external dependencies, you can place them in the vendor directory of your `.zip` file.

Vendor the Shim for Go Chaincodes

The Go chaincode shim dependency is no longer included with Hyperledger Fabric. The shim must now be vendored (imported) to Go chaincodes before they are installed on a peer.

You can use Go modules or a third-party tool such as govendor to vendor the chaincode shim and update it to the version that works with Hyperledger Fabric.

For more information, see Chaincode shim changes (Go chaincode only) and Upgrade Chaincodes with vendored shim in the Hyperledger Fabric documentation. For more information about Go modules, see Go Modules Reference.

Package a Node.js Chaincode

If you're writing a Node.js chaincode, you must create a `package.json` file with two sections:

- The `scripts` section declares how to launch the chaincode.
- The `dependencies` section specifies the dependencies.

The following is a sample `package.json` for a Node.js chaincode:

```
{
    "name": "chaincode_example02",
    "version": "1.0.0",
    "description": "chaincode_example02 chaincode implemented in Node.js",
    "engines": {
        "node": ">=8.4.0",
        "npm": ">=5.3.0"
    },
    "scripts": { "start" : "node chaincode_example02.js" },
    "engine-strict": true,
    "license": "Apache-2.0",
    "dependencies": {
        "fabric-shim": "~1.3.0"
    }
}
```

The following packaging rules apply to Node.js chaincodes.

- The `package.json` file must be in the root directory.
- The entry JavaScript file can be located anywhere in the package.
- If `"start" : "node <start>.js"` isn't specified in the `package.json` file, the `server.js` file must be in the root directory.

Compress the chaincode and package file in `.zip` format to install it on Oracle Blockchain Platform.

Package a Java Chaincode

If you're writing a Java chaincode, you can choose Gradle or Maven to build the chaincode.

If you're using Gradle, compress the chaincode, build.gradle, and settings.gradle to a file in `.zip` format to install it on Oracle Blockchain Platform. The following list shows sample files in a chaincode package:

```
Archive:  example_gradle.zip
 Length      Date    Time    Name
---------  ---------- -----   ----
      610  02-14-2019 01:36   build.gradle
       54  02-14-2019 01:28   settings.gradle
        0  02-14-2019 01:28   src/
        0  02-14-2019 01:28   src/main/
        0  02-14-2019 01:28   src/main/java/
        0  02-14-2019 01:28   src/main/java/org/
        0  02-14-2019 01:28   src/main/java/org/hyperledger/
        0  02-14-2019 01:28   src/main/java/org/hyperledger/fabric/
        0  02-14-2019 01:28   src/main/java/org/hyperledger/fabric/example/
     5357  02-14-2019 01:28   src/main/java/org/hyperledger/fabric/example/
SimpleChaincode.java
---------                     -------
     6021                     10 files
```

If you're using Maven, compress the chaincode and pom.xml to a file in `.zip` format to install it on Oracle Blockchain Platform. The following list shows sample files in a chaincode package:

```
Archive:  example_maven.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
     3313  02-14-2019 01:52   pom.xml
        0  02-14-2019 01:28   src/
        0  02-14-2019 01:28   src/chaincode/
        0  02-14-2019 01:28   src/chaincode/example/
     4281  02-14-2019 01:28   src/chaincode/example/SimpleChaincode.java
---------                     -------
     7594                     5 files
```

Testing a Chaincode

Test your chaincode after you write it. See the following topics:

- Use a Mock Shim to Test a Chaincode
- Deploy a Chaincode on a Peer to Test the Chaincode

Installing and Deploying a Chaincode

After you've tested your chaincode, you can deploy it by following the information in Typical Workflow to Deploy Chaincodes.

Upgrading a Chaincode

Upgrade a deployed chaincode by following the steps in Upgrade a Chaincode.

Use a Mock Shim to Test a Chaincode

This method of testing uses a mock version of the `shim.ChaincodeStubInterface` stub. With this you can simulate some functionality of your chaincode before deploying it to Oracle Blockchain Platform. You can also use this library to build unit tests for your chaincode.

1. Create a test file that matches the name of the chaincode file.

   For example, if `car_dealer.go` is the actual implementation code for you smart contract, you would create a test suite called `car_dealer_test.go` containing all the tests for `car_dealer.go`. Use the `*_test.go` format for the test suite file name.

2. Create your package and import statements.

```
package main

import (
    "fmt"
    "testing"

    "github.com/hyperledger/fabric/core/chaincode/shim"
)
```

3. Create your unit test.

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

    // Deploy mockStub using CarDemo as the target chaincode to unit test
    stub := shim.NewMockStub("mockStub", new(CarDemo))
    if stub == nil {
        t.Fatalf("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("001",
        [][]byte{[]byte("initVehiclePart"),
            []byte(serialNumber),
            []byte("tata"),
            []byte("1502688979"),
            []byte("airbag 2020"),
            []byte("aaimler ag / mercedes")})

    // We expect a shim.ok if all goes well
    if result.Status != shim.OK {
        t.Fatalf("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)
    }
}
```

> **Note**
>
> Not all interfaces of the stub are implemented. The following stub functions are not supported.
>
> - `GetQueryResult`
> - `GetHistoryForKey`
>
> Attempting to call either of these functions will result in an error.

Deploy a Chaincode on a Peer to Test the Chaincode

After you create a chaincode, you can install, deploy, and call it to test that it works correctly.

To learn more about writing a chaincode, see Write a Chaincode.

Complete these steps to deploy and test your chaincode.

1. Identify the channel or create a channel and add peers to it. See Join a Peer to a Channel.
2. Install the chaincode on the peers and deploy it on the channel. See Use Quick Deployment.
3. Use the `Invoke` and `query` REST APIs to test the chaincode with cURL through the REST proxy. See REST API for Oracle Blockchain Platform for descriptions of each endpoint and correct cURL syntax to start each operation.
4. Go to the Channels tab in the console and locate and click the name of the channel running the chaincode.
5. In the channel's **Ledger** pane, view the chaincode's ledger summary.

7

Deploy and Manage Chaincodes

This topic contains information to help you deploy, monitor, and upgrade chaincodes on the network.

Typical Workflow to Deploy Chaincodes

Complete the following tasks to deploy or upgrade chaincodes.

You must be an administrator to complete these tasks.

Task	Description	More Information
Use the wizard to fully or partially deploy a chaincode	For testing, use Quick Deployment to perform the deployment in one step, using default settings. For production, use Advanced Deployment to specify the deployment settings such as which peers to install the chaincode on and the endorsement policy you want to use.	Use Quick Deployment Use Advanced Deployment
Deploy a chaincode	Deploying a chaincode consists of approving and committing the chaincode definition.	Deploy a Chaincode Chaincode Life Cycle
Upgrade a chaincode	Upload a newer version of a chaincode package, or update a chaincode definition.	Upgrade a Chaincode

Use Quick Deployment

Use the quick deployment option to complete a one-step chaincode deployment. This option is recommended for chaincode testing.

The quick deployment uses default settings, installs the chaincode on all peers in the channel, deploys the chaincode using the default endorsement policy, and enables the chaincode in the REST proxy.

Note the following information:

- The process to deploy sample chaincodes is different than the process described in this topic. See Explore Oracle Blockchain Platform Using Samples.
- You can use the advanced deployment option to put your chaincode into production on the network. See Use Advanced Deployment.

You must be an administrator to complete this task.

1. Go to the console and select the Chaincodes tab.
2. In the Chaincodes tab, click **Deploy a New Chaincode**.

The Deploy Chaincode page is displayed.

3. Click **Quick Deployment**.

   The Deploy Chaincode (Quick) page is displayed.

4. In the **Package Label** field, enter a description of the chaincode package.

   Use the following guidelines when labeling the chaincode:

   - Use ASCII alphanumeric characters, dashes (-), and underscores (_).
   - The label must start and end only with ASCII alphanumeric characters. For example, you can't use labels such as `_mychaincode` or `mychaincode_`.
   - Dashes (-) and underscores (_) must be followed by ASCII alphanumeric characters. For example, you can't use names like `my--chaincode` or `my-_chaincode`.
   - The package label can be up to 50 characters long.

5. In the **Chaincode Type** list, select the language that the chaincode is written in. To deploy an external chaincode (chaincode as a service), select **External**. For more information about deploying chaincode as a service, see Deploy Chaincode from an External Service.

6. In the **Chaincode Name** field, enter a unique name for the chaincode. In the **Version** field enter a string value to specify the chaincode's version number.

   Use these guidelines when naming the chaincode:

   - Use ASCII alphanumeric characters, dashes (-), and underscores (_).
   - The name must start and end only with ASCII alphanumeric characters.
   - Dashes (-) and underscores (_) must be followed with ASCII alphanumeric characters.
   - The name and version can each be up to 64 characters long.
   - The chaincode version can also contain periods (.) and plus signs (+).

7. If the chaincode requires initialization, select **Init-required**.

   If **Init-required** is selected, the client application must invoke the `Init` function explicitly, by specifying the `isInit` flag, before calling any other function.

8. Review the other default settings and modify them as needed.

9. If you are deploying chaincode source in a `.zip` file, leave **Is Packaged Chaincode** deselected. If you are deploying a chaincode package in a `.tar.gz` file, select **Is Packaged Chaincode**.

10. Click **Upload Chaincode File** and browse for the chaincode file to upload and deploy.

11. Click **Submit**.

    The chaincode is installed on the channel's peers and deployed.

On the Channels tab, click the name of the channel that you deployed the chaincode to, and then click **Deployed Chaincodes**. The deployed chaincode's name, version, sequence number, and package ID are displayed in the summary table, as well as the approved and committed statuses.

Use Advanced Deployment

Use the advanced deployment option to specify the parameters required to deploy a chaincode into a production environment. For example, you'll specify which peers to install the chaincode on and the endorsement policy to use.

Note the following information:

- The process to deploy sample chaincodes is different than the process described in this topic. See Explore Oracle Blockchain Platform Using Samples.
- You can use the quick deployment option for chaincode testing. Quick deployment is a one-step deployment that uses default settings, installs the chaincode on all peers in the channel, and deploys the chaincode using a default endorsement policy. See Use Quick Deployment.

You must be an administrator to perform this task.

1. Go to the console and select the Chaincodes tab.
2. In the Chaincodes tab, click **Deploy a New Chaincode**.

   The Deploy Chaincode page is displayed.
3. Click **Advanced Deployment**.

   The Deploy Chaincode (Advanced) Step 1 of 2: Install page is displayed.
4. In the **Package Label** field, enter a description of the chaincode package.

   Use the following guidelines when labeling the chaincode:

   - Use ASCII alphanumeric characters, dashes (-), and underscores (_).
   - The label must start and end only with ASCII alphanumeric characters. For example, you can't use labels such as `_mychaincode` or `mychaincode_`.
   - Dashes (-) and underscores (_) must be followed by ASCII alphanumeric characters. For example, you can't use names like `my--chaincode` or `my-_chaincode`.
   - The package label can be up to 50 characters long.
5. In the **Chaincode Type** list, select the language that the chaincode is written in. To deploy an external chaincode (chaincode as a service), select **External**. For more information about deploying chaincode as a service, see Deploy Chaincode from an External Service.
6. In the **Target Peers** field, select one or more network peers to install the chaincode onto. To provide high availability, choose the appropriate number of peers from each partition. The peers you choose must be joined to the channel that you deploy the chaincode on.
7. If you are deploying chaincode source in a `.zip` file, leave **Is Packaged Chaincode** deselected. If you are deploying a chaincode package in a `.tar.gz` file, select **Is Packaged Chaincode**.
8. Click **Upload Chaincode File** and browse for the chaincode file to upload and deploy. Click **Next**.

   The chaincode is installed and the Deploy Chaincode (Advanced) Step 2 of 2: Deploy page is displayed.
9. Decide if you want to deploy the chaincode now or later.

   - Click **Close** to close the wizard and deploy later.
   - To deploy now, select the channel to deploy the chaincode on.

10. In the **Chaincode Name** field, enter a unique name for the chaincode. In the **Version** field enter a string value to specify the chaincode's version number.

    Use these guidelines when naming the chaincode:

    - Use ASCII alphanumeric characters, dashes (-), and underscores (_).
    - The name must start and end only with ASCII alphanumeric characters.
    - Dashes (-) and underscores (_) must be followed with ASCII alphanumeric characters.
    - The name and version can each be up to 64 characters long.
    - The chaincode version can also contain periods (.) and plus signs (+).

11. If the chaincode requires initialization, select **Init-required**.

    If **Init-required** is selected, the client application must invoke the `Init` function explicitly, by specifying the `isInit` flag, before calling any other function.

12. If required, enter an endorsement policy and private data collections, and then click **Next**. For more information about endorsement policies, see Specify an Endorsement Policy. For more information about private data collections, see Add Private Data Collections.

    Note the following information:

    - Deployment approves, commits, and initializes the chaincode on the channel.
    - If you do not change the endorsement policy, Oracle Blockchain Platform uses the default endorsement policy. The default endorsement policy is defined in the /Channel/Application/Endorsement policy of the channel where you are deploying the chaincode. The default endorsement policy gets an endorsement from any peer from any organization on the network.
    - When deployment is complete, the peers are able to accept chaincode invocations and can endorse transactions.

    The chaincode is deployed.

Deploy a Chaincode

To deploy a chaincode, it must be approved by organizations and then committed to a channel. After a chaincode is deployed, peers are able to accept chaincode invocations and can endorse transactions.

Note the following information:

- You must install the chaincode on the required peers before you can deploy it.
- You can deploy more than one chaincode on a channel.
- The process to deploy the sample chaincodes is different than the deployment process described in this topic. See Explore Oracle Blockchain Platform Using Samples.

You must be an administrator to complete this task.

1. Go to the console and click the **Chaincodes** tab.
2. On the Chaincodes page, locate the chaincode package and click its **More Actions** menu, and then select **Deploy**.

   The Deploy Chaincode dialog is displayed.
3. Enter information about where and how to deploy the chaincode.

Field	Description
Channel	Select the channel for the chaincode to run on.
Chaincode Type	Select the language that the chaincode is written in. For external chaincodes (chaincode as a service), select **External**.
Chaincode Name	Enter a unique name, up to 64 characters long, for the deployed chaincode. • Use ASCII alphanumeric characters, dashes (-), and underscores (_). • The name must start and end only with ASCII alphanumeric characters. • Dashes (-) and underscores (_) must be followed with ASCII alphanumeric characters.
Version	Enter a string value, up to 64 characters long, to specify the chaincode's version number. • Use ASCII alphanumeric characters, dashes (-), underscores (_), periods (.) and plus signs (+).
Init-required	Select if the chaincode requires initialization. If selected, the client application must invoke the `Init` function explicitly, by specifying the `isInit` flag, before calling any other function.
Endorsement Policy	In this section, specify the policy required to endorse the chaincode. If you don't specify an endorsement policy, then the default endorsement policy is used. The default endorsement policy gets an endorsement from any peer on the network.
Private Data Collection	In this section, add one or more private data collections. Private data collections specify subsets of organizations that endorse, commit, or query private data on the channel you deploy the chaincode on.

4. Click **Deploy**.

   The chaincode is deployed.

5. To confirm that the chaincode was deployed, go to the Channels page and click the name of the channel that you deployed the chaincode on. Go to the Deployed Chaincodes page and confirm that the chaincode is listed in the summary table.

Deploy Chaincode from an External Service

You can run external chaincode, or chaincode as a service, on Oracle Blockchain Platform.

You can run chaincode as a service that is managed externally instead of being built and launched on a peer node. This functionality decouples creating the chaincode from deploying it to the Hyperledger Fabric network. Instead, the chaincode can be managed by an administrator independently of the peer node.

1. Create a `connection.json` file with the address information of the external host, and then compress the file in `.zip` format. The following text shows a sample `connection.json` file.

   ```
   {
   "address": "example.com:9999",
   "dial_timeout": "10s",
   "tls_required": false
   }
   ```

   Use the public IP address or host name and the public port for the address when you specify the remote host or VM where the chaincode is managed.

2. When you deploy chaincode, select **External** for the **Chaincode Type**. For **Chaincode Source**, upload the `.zip` file that you created in the previous step.

Chaincode Life Cycle

The chaincode life cycle describes the process of installing chaincode on peers and deploying it on a channel.

The chaincode life cycle is based on the capabilities of the Hyperledger Fabric platform, which allows for the decentralized governance of chaincodes. Multiple organizations can agree on chaincode parameters, including the chaincode endorsement policy, before a chaincode can interact with the ledger. These functions are implemented in the new quick deployment and advanced deployment options, as well as in the REST API. For more information about the life cycle, see Fabric chaincode lifecycle in the Hyperledger Fabric documentation.

Typically, to deploy an installed chaincode, you use quick deployment or advanced deployment in the console. The deployment process includes packaging and installing the chaincode as well as approving and committing the chaincode definition. You can also use the REST API to complete the approval and commitment operations separately.

Package and Install a Chaincode

When you install chaincode in Oracle Blockchain Platform, the chaincode is packaged, installed, and a package ID is generated automatically. The package ID is displayed on the **Chaincodes** tab of the console.

Approve a Chaincode Definition

Before a chaincode can be deployed to a channel, the chaincode definition must be approved by enough organizations to satisfy the LifecycleEndorsement policy of the channel. The default LifecycleEndorsement policy in Oracle Blockchain Platform lets any organization approve the chaincode definition (as opposed to a majority of organizations). The chaincode definition includes the following parameters, which must be the same for all organizations: `Chaincode Name`, `Version`, `Sequence`, `Endorsement Policy`, `Private Data Collection`, and `Init-required`. A chaincode definition can also include a `Package ID`, which does not have to be the same for all organizations.

After a chaincode definition is approved, one organization can collect endorsements from peers of the approving organizations and then commit the chaincode definition to the channel.

To approve a chaincode definition by using the REST API, see Approve a Chaincode Definition in a Channel.

In the console, when you use quick deployment or advanced deployment the approval and commitment steps are both attempted.

Commit a Chaincode Definition

To commit an approved chaincode definition by using the REST API, see Commit a Chaincode Definition in a Channel.

In the console, you can see chaincode definitions that are approved but not committed on the Deployed Chaincodes page for the channel. You can use the **More Actions** menu to commit the approved chaincode.

Chaincode Life Cycle Scenarios

Scenario	Description
Join a channel	Typically in the console you do not approve a chaincode definition without then committing it. If you join a shared channel where a chaincode definition was committed by another organization, you will see the chaincode definition listed as committed but not approved on the Deployed Chaincodes page for the channel. You can use the **More Actions** menu to approve the chaincode definition and also to associate a package ID. You do not need to commit the package definition again.
Update an endorsement policy	You can update the endorsement policy in the chaincode definition without reinstalling the chaincode. On the Deployed Chaincodes page for the channel, use the More Actions menu to upgrade the chaincode definition. Expand **Endorsement Policy** and specify a new policy, then click **Upgrade**.
Approve a definition without installing	In a multiple organization scenario, to approve a chaincode definition without installing the chaincode package, do not specify a package ID. You endorse the definition of the chaincode that is committed to the channel, but the chaincode is not installed on peers in your organization. You will not be able to use the chaincode to endorse transactions or query the ledger.
Disagreement on definitions	In a multiple organization scenario, an organization that doesn't approve a chaincode definition or approves a different chaincode definition is not able to run the chaincode on their peers. If other organizations get enough endorsements to commit the definition to the channel, those organizations can use the chaincode. Transactions are still added to the ledger on the peers of all organizations. If organizations do not agree on a chaincode definition and no organizations get enough endorsements to commit the definition to the channel, the definition cannot be committed and therefore the chaincode cannot run.
Multiple organizations install different packages	You can specify a different package ID when you approve a chaincode definition for a channel with multiple organizations. If the definition name and endorsement policy are the same, then channel members can install chaincode that is specific to their organization, but which reads and writes data to the same chaincode namespace.
Create multiple chaincodes from one package	Similarly, you can approve and commit the same chaincode package multiple times, specifying a different name for each definition. Multiple instances of the chaincode run on the channel. If you also specify a different endorsement policy for each definition, then each chaincode instance is subject to a different endorsement policy.

Specify an Endorsement Policy

You can add an endorsement policy when you deploy a chaincode. An endorsement policy specifies the members with peers that must approve, or properly endorse, a chaincode transaction before it's added to a block and submitted to the ledger.

Endorsement guarantees the legitimacy of a transaction. When you deploy a chaincode on a channel, you can specify an endorsement policy. If you don't specify an endorsement policy, then the default endorsement policy is used. The default endorsement policy gets an endorsement from any peer on the network.

A member's endorsing peers must have ReaderWriter permissions on the channel. When a transaction is processed, each endorsing peer returns a signed read-write set. After the client has enough endorsements to meet the endorsement policy requirements, then the client bundles the common read-write set with the signature from the endorsing peers and sends everything to the ordering service, which orders and commits the transactions into blocks and then to the ledger.

You can go to the Channels page to view a deployed chaincode's endorsement policy. See View an Endorsement Policy. You can't modify a deployed chaincode's endorsement policy. If you need to change an endorsement policy, then you must redeploy the chaincode or upgrade it to another version and specify a different endorsement policy.

You must be an administrator to complete this task.

1. Go to the console and click the **Chaincodes** tab.
2. Locate the chaincode package that you want to deploy and use the **More Actions** menu to begin the deployment process.
3. On the Deploy Chaincode window, expand **Endorsement Policy**.
4. Select **Default**, **Signature Policy** or **Channel Config Policy**, and then specify an expression for the endorsement policy.

   For more information about endorsement policies, see Endorsement policies in the Hyperledger Fabric documentation.
5. Complete the other fields on the Deploy Chaincode page as needed.
6. Click **Deploy**.

View an Endorsement Policy

You can view a deployed chaincode's endorsement policy.

You might need to view a deployed chaincode's endorsement policy to see how it was set up, how you need to choose transaction endorsers based on the policy, or to help resolve an endorsement failure.

You can't modify the endorsement policy for a deployed chaincode. If you need to change an endorsement policy, then you must redeploy the chaincode or upgrade it to another version and specify a different endorsement policy.

1. Go to the console and click the **Channels** tab.
2. Click the name of the channel where the chaincode is deployed, and then click **Deployed Chaincodes**.
3. In the table, click the **More Actions** menu icon for the chaincode, and then click **View Chaincode Definition**.

The Chaincode Definition window is displayed.

4. Expand **Endorsement Policy**.

   The chaincode endorsement policy is displayed.

Find Information About Chaincodes

You can find information about the chaincodes in your network, including how many peers the chaincode is installed on and if the chaincode has been deployed. You can view information about chaincode packages and chaincode definitions.

1. Go to the console and click the **Chaincodes** tab.

   The Chaincodes page is displayed. The chaincode table lists the chaincode packages that are available on the network, with information about how many peers a package is installed on and how many channels a package is deployed on.

2. In the table, click a chaincode package to see more information about which peers it's installed on, and its names and versions for the channels it's deployed on.

   - When you stop a peer node, Oracle Blockchain Platform removes the peer's listing on the Chaincodes page.
   - If you stop all peers that have the chaincode installed, then the Chaincodes page doesn't list the chaincode. To list the chaincode, start at least one peer node that has the chaincode installed on it.
   - Use the **More Actions** menu icon to deploy the chaincode package to a different channel, or to the same channel but with a different chaincode definition. You can also download the chaincode package and delete the chaincode package. You might delete a chaincode package to free up space for installing other chaincodes. When you delete a chaincode package, it is not recoverable.

3. To see the definitions of deployed chaincodes, click the **Channels** tab.

4. Click the name of the channel where the chaincode is deployed, and then click **Deployed Chaincodes**.

   In the table, you can use the **More Actions** menu to get information about a chaincode definition or to upgrade a chaincode.

Delete a Chaincode

You can delete obsolete or unused chaincode packages to free up disk space.

You might delete a chaincode package to free up space for installing other chaincodes. When you delete a chaincode package, it is not recoverable.

1. Go to the console and select the **Chaincodes** tab.

   The Chaincodes page is displayed. The chaincode table lists the chaincode packages that are available on the network, with information about how many peers a package is installed on and how many channels a package is deployed on.

2. In the table, click the **More Actions** menu item for the chaincode to delete, and then click **Delete**.

3. Click **Yes** to confirm the chaincode deletion.

Manage Chaincode Versions

Each chaincode that you deploy or upgrade consists of a chaincode package and a chaincode definition.

1. Go to the console and click the **Channels** tab.
2. Select the channel that you want to inspect and then click **Deployed Chaincodes**.

   The deployed chaincodes summary table lists the names and versions of the chaincodes deployed to the channel.
3. Click the **More Actions** menu icon for a chaincode and then click **View Chaincode Definition** to see the chaincode definition, including the endorsement policy and private data collections.
4. Click a package ID. The Installed Peers Summary page is displayed, showing which peers the package is installed on. You can click the peer to view more information about it.
5. Click the Deployed on Channels pane to see all of the channels the chaincode is deployed on. You can click a channel to view more information about it.

   From this pane, you can click Deploy on a New Channel to deploy the chaincode package to a different channel, or to the same channel using a different chaincode definition.

Upgrade a Chaincode

If a developer modifies a chaincode's source, then you'll need to deploy it to a new version of the chaincode.

You can deploy different versions of the same chaincode on different channels.

You must be an administrator to perform this task. If you use the console, the upgrade process includes both approving and committing the upgraded chaincode. You can also use the REST API to upgrade a deployed chaincode by using the same calls that you use to install, approve, and commit a chaincode. For more information, see REST API for Oracle Blockchain Platform on Oracle Cloud Infrastructure (Gen 2).

1. Go to the console and click the **Channels** tab.

   The Channels page is displayed and the table lists all of the channels on the network.
2. Click the channel where the chaincode that you want to upgrade is deployed, and then click **Deployed Chaincodes**.
3. Locate the chaincode that you want to upgrade, click **More Actions**, and select **Upgrade**.

   The Upgrade Chaincode page is displayed.
4. Specify a **Chaincode Version** and select a **Package ID** to use in the chaincode definition.
5. If the chaincode requires initialization, select **Init-required**.

   If **Init-required** is selected, the client application must invoke the `Init` function explicitly, by specifying the `isInit` flag, before calling any other function.
6. If required, enter an endorsement policy and private data collections, and then click **Upgrade**.

   The chaincode is upgraded and deployed.

What Are Private Data Collections?

Private data collections specify subsets of organizations that endorse, commit, or query private data on the channel.

Use private data collections in cases where you want a group of organizations on the channel to share data and to prevent the other organizations on the channel from seeing the data. Private data is distributed peer to peer and not by blocks, so the transaction data is kept confidential from the ordering service. Collections help you reduce the number of channels and their required maintenance on your network.

The primary components in a private data collection are:

- The private data that you specify in your private data collection definition. Private data is sent with the gossip protocol from peer to peer within the organizations that you specify in your policy. Private data is stored in a private database on the peer. The ordering service isn't used and can't see the private data.
- A hash of the data, which is endorsed, ordered, and written to each peer on the channel. This hash is evidence of the transaction and can be used for audit purposes.

When you deploy a chaincode, you can associate it with one or more private data collections.

Add Private Data Collections

You can add private data collections to channels. Private data collections specify subsets of organizations that endorse, commit, or query private data on the channel.

Use private data collections in cases where you want a group of organizations on the channel to share data within a transaction and to prevent the other organizations on the channel from seeing the data.

If you're going to use private data collections across the organizations in your network, you must configure anchor peers. Anchor peers facilitate private data gossip among the organizations. See Add an Anchor Peer.

You specify the private data collections when you deploy the chaincode.

1. Go to the console and click the **Chaincodes** tab.
2. Locate the chaincode that you want to deploy and begin the deployment process.
3. Expand the Private Data Collections section and add the collection definition as needed.

Field	Description
Collection Name	Enter the collection's name. You'll reference this name in the chaincode.

Field	Description
Policy	Create the policy to specify which organizations are included in the collection and which peers can store the private data. Each member listed in the policy must be included in an `OR` signature policy list. To support read/write transactions, the private data distribution policy must contain more organizations than the chaincode endorsement policy because peers must have the private data to endorse transactions. For example, in a channel with ten organizations, five of the organizations are included in a private data collection policy, but the endorsement policy requires three organizations to endorse a transaction.
Peers Required	Enter the number of peers that each endorsing peer must distribute private data to before the peer signs the endorsement and returns the proposal response. Set this value to 1 or more peers to ensure the following: • Redundancy of the private data on multiple peers in the network. • Availability of the private data if the endorsing peers become unavailable. Note that setting this value to 0 means that distribution isn't required. However, if the **Max Peer Count** field is set to greater than 0, private data distribution might still occur.
Max Peer Count	Enter the maximum number of peers that the current endorsing peer attempts to distribute the data to. This is to ensure redundancy so that peers are available between endorsement time and commit time to pull the private data if an endorsing peer isn't available. If you set this value to 0, the private data isn't distributed at the time of endorsement. This causes private data pulls against the endorsing peers on all authorized peers at commit time.
Block to Live	Enter the length in number of blocks that you want data to reside on the private database. The data is purged when the number of blocks is reached. Set this value to 0 if you never want to purge the data. Note that a peer can fail to pull private data from another peer if a private data collection's `blocktolive` value is less than 10, and its `requiredPeerCount` and `maxPeerCount` values are less than the total number of peers in the channel. This is a known Hyperledger Fabric issue.

Field	Description
Endorsement Policy	Optionally, specify an endorsement policy for the collection that overrides the chaincode's endorsement policy. Choose a **Policy Type** of either **Signature Policy** or **Channel Config policy** to use a signature policy or an existing channel configuration policy. For **Policy**, specify an expression that represents the endorsement policy. For more information, see Endorsement policies in the Hyperledger Fabric documentation.
Member Only Read	Select to automatically prevent members of organizations that are not part of the collection from reading private data.
Member Only Write	Select to automatically prevent members of organizations that are not part of the collection from writing private data.

4. Click **Add New Collection**. Your collection's information is displayed in the private data collection table.
5. If needed, specify other collections.
6. Complete the other fields on the Deploy Chaincode page as needed.
7. Click **Deploy**.

View Private Data Collections

You can view information about a chaincode's private data collections.

After you deploy a chaincode, you might need to view its private data collections to see how they were defined.

You can’t modify the private data collections for a deployed chaincode. To change the private data collections, upgrade the chaincode and specify new private data collections.

1. Go to the console and select the **Channels** tab.
2. Click the name of the channel where the chaincode is deployed, and then click **Deployed Chaincodes** to open the Deployed Chaincodes Summary page.
3. Click the **More Actions** menu icon for the deployed chaincode.
4. Click **View Chaincode Definition**.
5. On the Chaincode Definition window, expand **Private Data Collection**, and then locate the collection that you want to view.

8

Develop Blockchain Applications

Blockchains require smart contracts (chaincode) to update the ledger. In addition, you will also require a client application that utilizes either the Oracle Blockchain Platform REST API or native Hyperledger Fabric SDK to interact with the blockchain directly. There are other operational and administrative tasks to consider, namely the creation of peers and channels and installation of chaincode.

Topics

Before You Develop an Application

Before you write an application, download and use the sample applications, and ensure that you've the correct certificates and privileges to run an application.

Oracle Blockchain Platform provides downloadable samples that help you understand how to write chaincodes and applications. See:

- What Are Chaincode Samples?
- Explore Oracle Blockchain Platform Using Samples

Oracle Blockchain Platform uses Hyperledger Fabric as its foundation. Use the Hyperledger Fabric documentation to help you write applications. Read the *Key Concepts* and *Tutorials* sections before you write your own application: Hyperledger Fabric documentation.

Prerequisites for Application Development

A user ID and password for the application user must exist in Oracle Identity Cloud Service. Depending on the functions in the application, this user must have the following prerequisites:

- To install and deploy chaincode:
  - You must have administrative access in order to install or deploy chaincode.
  - You must export the admincerts, cacerts, and tlscacerts certificates as described in Export Certificates so that they can be placed in your application in the peer and orderer nodes crypto folders.
  - You must export the admin credentials similarly to how you exported the certificates (from the action menu, select **Export Admin Credential**). This will download a ZIP file containing the signed certificate and keystore files that need to be placed in your application in the peer and orderer nodes crypto folders.
- To run operations against an installed and deployed chaincode:

  - You must export the admincerts, cacerts, and tlscacerts certificates as described in Export Certificates so that they can be placed in your application in the peer node crypto folders.
  - You must export the tlscacerts certificate for the orderer node as described in Join the Participant or Scaled-Out OSNs to the Founder's Ordering Service so that it can be placed in your application.
  - The chaincode you're invoking must be installed and deployed to a channel and node that your user ID has access to.
  - A REST proxy node must be configured and the chaincode enabled for REST proxy access. The user ID and password for the node must be provided.
- To run functions against a REST API endpoint:
  - The chaincode you're invoking must be installed and deployed to a channel and node that your user ID has access to.
  - A REST proxy node must be configured and the chaincode enabled for REST proxy access. The user ID and password for the node must be provided.

Use the Fabric Gateway to Develop Applications

Applications can use the Fabric Gateway client API to run queries and updates on the ledger.

You can install and use the Fabric Gateway client API to develop applications for Oracle Blockchain Platform. Fabric Gateway is available for Go, Node.js, and Java.

The REST APIs provided by Oracle Blockchain Platform have been created with maximum flexibility in mind; you can invoke a transaction, invoke a query, or view the status of a transaction. See REST API for Oracle Blockchain Platform.

However, this flexibility means that you'll likely want to wrap the existing API endpoints in an application to provide object-level control. Applications can contain much more fine-grained operations.

Previous versions of Hyperledger Fabric supported the Hyperledger Fabric SDK. Hyperledger Fabric SDK is now deprecated. To learn more about Fabric Gateway, including information on migrating existing applications from Hyperledger Fabric client SDKs to the Fabric Gateway client API, see Fabric Gateway.

Use the Hyperledger Fabric SDKs to Develop Applications

The Hyperledger Fabric SDKs are now deprecated. However, you can install and use the Hyperledger Fabric SDKs to develop applications for Oracle Blockchain Platform.

The REST APIs provided by Oracle Blockchain Platform have been created with maximum flexibility in mind; you can invoke a transaction, invoke a query, or view the status of a transaction. See REST API for Oracle Blockchain Platform.

However, this flexibility means that you'll likely want to wrap the existing API endpoints in an application to provide object-level control. Applications can contain much more fine-grained operations.

SDK Versions

Multiple versions of the Hyperledger Fabric SDKs are available. Use the version of the SDK that matches the version of Hyperledger Fabric that your instance is based on.

Installing the Hyperledger Fabric SDK for Node.js

Information about how to use the Fabric SDK for Node.js can be found here: Hyperledger Fabric SDK for Node.js documentation

On the **Developer Tools** tab, open the **Application Development** pane. You can install the Hyperledger Fabric Node.js SDK by using the link on this tab.

Installing the Hyperledger Fabric SDK for Java

Information about how to use the Fabric SDK for Java can be found here: Hyperledger Fabric SDK for Java documentation

On the **Developer Tools** tab, open the **Application Development** pane.

- You can install the Hyperledger Fabric Java SDK by using the link on this tab.
- (Hyperledger Fabric v2.x) You must modify the SDK to work with Oracle Blockchain Platform by following the instructions in Update the Hyperledger Fabric Java SDK to Work with Oracle Blockchain Platform.

Install a build tool such as Apache Maven.

Structuring your Application

Structure your Java application similar to the following example:

```
/Application
  /artifacts
    /cypto
      /orderer
        Contains the certificates required for the application to act on the
orderer node
        In participant instances only contains TLS certificates
      /peer
        Contains the certificates required for the application to act on the
peer node
    /src
      chaincode.go if installing and deploying chaincode to the blockchain
  /java
    pom.xml or other build configuration files
    /resources
      Any resources used by the Java code, including artifacts such as the
endorsement policy yaml file and blockchain configuration properties
    /src
      Java source files
```

Structure your Node.js application similar to the following example:

```
/Application
  /artifacts
    /cypto
      /orderer
        Contains the certificates required for the application to act on the
orderer node
        In participant instances only contains TLS certificates
      /peer
        Contains the certificates required for the application to act on the
peer node
    /src
      chaincode.go if installing and deploying chaincode to the blockchain
```

```
/node
  package.json file
  application.js
  /app
    Any javascript files called by the application
    /tools
```

Running the application

You're now ready to run and test the application. In addition to any status messages returned by your application, you can check the ledger in the Oracle Blockchain Platform console to see your changes:

1. Go to the **Channels** tab in the console, and then locate and click the name of the channel running the chaincode.
2. In the channel's Ledger pane, view the chaincode's ledger summary.

Update the Hyperledger Fabric Go SDK to Work with Oracle Blockchain Platform

You must update the `NormalizeURL` function in the Hyperledger Fabric Go SDK to use it with the Kubernetes-based version of Oracle Blockchain Platform.

1. Open the `client.go` file in the Hyperledger Fabric Go SDK for editing: `vendor/github.com/hyperledger/fabric-sdk-go/internal/github.com/hyperledger/fabric-ca/lib/client.go`
2. Update the `NormalizeURL` function as shown in the following code.
   `NormalizeURL` function before editing:

```
// NormalizeURL normalizes a URL (from cfssl)
func NormalizeURL(addr string) (*url.URL, error) {
    addr = strings.TrimSpace(addr)
    u, err := url.Parse(addr)
    if err != nil {
        return nil, err
    }
    if u.Opaque != "" {
        u.Host = net.JoinHostPort(u.Scheme, u.Opaque)
        u.Opaque = ""
    } else if u.Path != "" && !strings.Contains(u.Path, ":") {
        u.Host = net.JoinHostPort(u.Path, "")
        u.Path = ""
    } else if u.Scheme == "" {
        u.Host = u.Path
        u.Path = ""
    }
    if u.Scheme != "https" {
        u.Scheme = "http"
    }
.
.
.
```

`NormalizeURL` function after editing:

```
// NormalizeURL normalizes a URL (from cfssl)
func NormalizeURL(addr string) (*url.URL, error) {
    addr = strings.TrimSpace(addr)
    u, err := url.Parse(addr)
    if err != nil {
        return nil, err
    }
    if u.Opaque != "" {
        u.Host = net.JoinHostPort(u.Scheme, u.Opaque)
        u.Opaque = ""
    } else if u.Host != "" && !strings.Contains(u.Host, ":") {
        u.Host = net.JoinHostPort(u.Host, "")
        u.Path = ""
    } else if u.Scheme == "" {
        u.Host = u.Path
        u.Path = ""
    }
.
.
.
```

3. Save the `client.go` file.

Update the Hyperledger Fabric Java SDK to Work with Oracle Blockchain Platform

There's an incompatibility between an OCI infrastructure component and the Java SDK provided with Hyperledger Fabric v2.x. You must update the SDK to use it with Oracle Blockchain Platform.

Methods of updating the Hyperledger Fabric SDK

There are two ways of updating the SDK:

- Download the modified package from the console. We've created an updated `grpc-netty-shaded-1.38.0.jar` file, which is the module referenced by the Java SDK that requires modifications.
- Build the package manually, as described in the following steps.

To download the `grpc-netty-shaded-1.38.0.jar` file, click the console's **Developer Tools** tab, and then select the **Application Development** pane.

Manually building the package

For the `fabric-sdk-java` project, complete the following steps to rebuild the `grpc-netty-shaded` package to connect the peers and orderers with the `grpcs` client (via TLS). The `grpc-netty-shaded` package is a sub-project of `grpc-java`.

1. Install project dependencies:

```
mvn install
```

2. Download the `grpc-java` source code:

   ```
   git clone https://github.com/grpc/grpc-java.git
   ```

3. Find the `grpc` version that your `fabric-sdk-java` project uses and check out the code. In the `grpc-java` directory, check out the version of `grpc` that the `fabric-sdk-java` project uses:

   ```
   cd grpc-java && git checkout v1.38.0
   ```

4. Change the code to avoid an `alpn` error from the server side. Create a `grpc-java-patch` patch file with the following contents:

   ```
   diff --git a/netty/src/main/java/io/grpc/netty/ProtocolNegotiators.java b/
   netty/src/main/java/io/grpc/netty/ProtocolNegotiators.java
   index 19d3e01b7..ebc4786a3 100644
   – a/netty/src/main/java/io/grpc/netty/ProtocolNegotiators.java
   +++ b/netty/src/main/java/io/grpc/netty/ProtocolNegotiators.java
   @@ -611,7 +611,8 @@ final class ProtocolNegotiators {
   SslHandshakeCompletionEvent handshakeEvent = (SslHandshakeCompletionEvent)
   evt;
   if (handshakeEvent.isSuccess()) {
   SslHandler handler = ctx.pipeline().get(SslHandler.class);

       if (sslContext.applicationProtocolNegotiator().protocols()
       + if (handler.applicationProtocol() == null
       + || sslContext.applicationProtocolNegotiator().protocols()
       .contains(handler.applicationProtocol())) {
       // Successfully negotiated the protocol.
       logSslEngineDetails(Level.FINER, ctx, "TLS negotiation succeeded.",
   null);
   ```

5. Apply the patch:

   ```
   git apply grpc-java.patch
   ```

6. Build the project to generate the target patched package. Use gradle to build the `grpc-java-shaded` project:

   ```
   cd netty/shaded
   gradle build -PskipAndroid=true -PskipCodegen=true
   ```

   After the build completes, the target patched `.jar` package is available at `grpc-java/netty/build/libs/grpc-netty-shaded-1.38.0.jar`.

7. Add the patched package into your Maven local repository.

   Replace the original `grpc-netty-shaded .jar` package with the patched package in either of the following two ways:

   - Use Maven to install the package by local file:

     ```
     mvn install:install-file -Dfile=grpc-netty-shaded-build/grpc-netty-
     shaded-1.38.0.jar -DgroupId=io.grpc -DartifactId=grpc-netty-shaded -
     Dversion=1.38.0 -Dpackaging=jar
     ```

You must keep the target `groupid`, `artifactid`, and `version` the same as the package you want to replace.

- Manually replace your package. Go to the local Maven repository, find the directory where the target package is located, and replace the package with patched package.

8. Run the project.

Use the REST APIs to Develop Applications

The REST APIs provided by Oracle Blockchain Platform have been created with maximum flexibility in mind; you can call a transaction, run a query, or view the status of a transaction. However, this means that you'll likely want to wrap the existing API endpoints in an application to provide object-level control. Applications can support much more fine-grained operations.

Any application that uses the REST APIs requires the following information:

- The chaincode name and version.
- The REST server URL and port, and the user ID and password for the REST node.
- Functions to call transactions against or query the ledger.

See REST API for Oracle Blockchain Platform for information on the existing operations, including examples and usage syntax.

Structuring your Application

Structure your REST API application similar to the following example:

```
/Application
  /artifacts
    /crypto
      /orderer
        Contains the certificates required for the application to act on the
orderer node
        In participant instances only contains TLS certificates
      /peer
        Contains the certificates required for the application to act on the
peer node
    /src
  /REST
    Application script containing REST API calls
```

Make Atomic Updates Across Chaincodes and Channels

You can use atomic transactions to complete multiple transactions across channels and chaincodes in an atomic manner.

An atomic transaction is an indivisible series of data operations that either all succeed, or none succeed.

Atomic transactions can be useful in complex situations where multiple chaincodes are deployed to separate channels. You can use atomic transactions to maintain data consistency while running multiple blockchain transactions, even if a network or system failure occurs. Oracle Blockchain Platform supports atomic transactions by using the two-phase commit protocol, where an initial phase where each data operation is prepared is followed by a phase where each data operation is actually committed.

Atomic transactions work at the application level. Typically you do not need to change existing chaincode logic to support atomic transactions. Because one or more additional arguments are added by the atomic transactions framework, make sure that any existing chaincode does not perform strict checks on the number of arguments passed in the chaincode method. Atomic transactions are supported by the following REST API endpoint:

- `restproxy/api/v2/atomicTransactions`

The REST API endpoint prepares the transactions as defined by your chaincode, and then uses built-in chaincode functions to either to commit all of the transactions, or to roll back all of the transactions if there are any errors during the prepare phase. For more information about the REST endpoints to use to implement atomic transactions, see Atomic Transactions REST Endpoints.

Each atomic transaction is composed of two or more blockchain transactions. The result (the `returnCode` value) of the atomic transaction is either `Success` or `Failure`. In an atomic transaction, each requested blockchain transaction is split into two distinct operations: a prepare phase and then either a commit or a rollback phase.

- In the prepare phase, each transaction is endorsed as usual, but instead of being finalized, the changes are staged and the values are locked to prevent other transactions from modifying the staged values.
- If the prepare phase is successful for each blockchain transaction, then the transactions are endorsed and committed by using built-in chaincode. The previously locked values are unlocked, and the result of the atomic transaction is `Success`.
- If the prepare phase fails for any blockchain transaction, then all other transactions where the prepare phase succeeded are rolled back, again by using built-in chaincode. The staged changes are removed and the previously locked values are unlocked. The result of the atomic transaction is `Failure`.

Because the atomic transactions works by locking keys, you might receive a `Two_Phase_Commit_Lock` error if a different transactions attempts to modify a key that is locked by a currently active atomic transaction that is prepared. This can occur in one of the following two scenarios:

- An atomic transaction is still in the prepared phase, and a different transaction attempts to modify a key that was locked by the prepared transaction. In this case, the system is working as designed. If you encounter this error, retry the second transaction. This is analogous to how applications handle phantom read errors or multi-version concurrency control (MVCC) errors.
- The `GlobalStatus` value returned by the atomic transaction is `HeuristicOutcome`. In this case, an atomic transaction operation was canceled because one of the commit operations failed. This is a rare occurrence and might need to be resolved manually. One side effect of a heuristic outcome is that some keys may be left locked by transactions which failed to be committed or rolled back. In this case, use the following REST API endpoint to unlock the atomic transaction:
  - `restproxy/api/v2/atomicTransactions/{globalTransactionId}`

  For more information about the REST endpoint to use to unlock atomic transactions, see Unlock Atomic Transaction.

Scenario: Explore Atomic Transactions Using Samples

Consider the following example, which uses two of the sample chaincodes that are included with Oracle Blockchain Platform, Balance Transfer and Marbles. The Balance Transfer sample represents two parties with the ability to transfer funds between account balances. The Marbles sample lets you create marbles and exchange them between owners. You could use

individual (non-atomic) transactions to buy a marble by exchanging funds in the Balance Transfer chaincode and changing the ownership of the marble in the Marbles chaincode. However, if an error occurs with one of those transactions, the ledger might be left in an inconsistent state: either the funds were transferred but not the marble or the marble is transferred but not paid for.

In this scenario, you can use the existing chaincode with the REST API endpoints that support atomic transactions. The exchange of funds and the transfer of ownership of the marble must both succeed or both fail. If either transaction encounters an error, then neither transaction is committed. To explore this scenario, complete the following steps:

1. Install the Balance Transfer and Marbles samples on different channels. For more information on installing the samples, see Explore Oracle Blockchain Platform Using Samples.
2. In the Marbles sample, invoke the `Create a new marble` action to create a number of marbles for various marble owners.
3. Use the `Invoke Atomic Transaction` REST endpoint to complete atomic transactions that invoke both the Marbles and the Balance Transfer samples.

For example, the following transaction transfers a marble named `marble1` to Tom, and sends 50 coins from account a to account b.

```
{
 "transactions": [
   {"chaincode":"obcs-marbles","args":["transferMarble", "marble1",
"tom"],"timeout":0, "channel":"goods"},
   {"chaincode":"obcs-example02","args":["invoke", "a", "b",
"50"],"timeout":0, "channel":"wallet"}
 ],
 "isolationLevel": "serializable",
 "prepareTimeout": 10000,
 "sync": true
}
```

In the previous transaction, if both transactions succeed in the prepare phase, then both transactions are committed to the ledger. If there is an error with either transaction, then neither transaction is committed during the second phase. Instead, both transactions are rolled back. For example, if there are less than 50 coins in account a, then no money is taken from the account and no marble is transferred to Tom.

Ethereum Interoperability

You can include Ethereum-based transactions in an atomic transaction workflow.

Growing use of public blockchains and tokenization capabilities across both public and permissioned blockchains drives the need for their interoperability. Common scenarios include asset exchange across different ledgers, business transactions on permissioned blockchains that are linked to cryptocurrency payments on public chains, publishing proof of a permissioned blockchain transaction on a public blockchain, and so on. To enable interoperability for these and other scenarios, Oracle Blockchain Platform provides interoperability with Ethereum and with any EVM-based networks that support standard web3 protocols. The interoperability function works by incorporating the Geth Ethereum client in the REST proxy and enabling it to orchestrate an optimized two-phase commit protocol that includes both Oracle Blockchain Platform and Ethereum/EVM transactions through a single REST API called `atomicTransactions`. You can use the `atomicTransactions` API to send

multiple chaincode transactions for multiple Oracle Blockchain Platform channels, and can optionally add an Ethereum transaction that will run atomically with the Oracle Blockchain Platform transactions.

Unlike Oracle Blockchain Platform transactions, Ethereum transactions cannot be broken down into the prepare and commit phases of the two-phase commit protocol. To include Ethereum transactions as part of an atomic workflow, Oracle Blockchain Platform uses a last resource commit (LRC) optimization. After all of the Oracle Blockchain Platform transactions are in the prepared state, the Ethereum transaction is started. If the Ethereum transaction succeeds, then the Oracle Blockchain Platform transactions are committed. If the Ethereum transaction fails, then the Oracle Blockchain Platform transactions are rolled back.

Ethereum transactions have a concept of **finality**. An Ethereum transaction can run successfully but it does not achieve finality until it's part of a block that can't change. You can use the `finalityParams` parameters to control whether to check for finality and how long to wait for it, either in blocks or in seconds. Typically, if you wait for six blocks to be generated on the public Ethereum blockchain network (**Mainnet**), you can assume that transaction finality was achieved. In private Ethereum networks, typically you do not need to wait as long for finality.

Transferring an NFT to an Ethereum network

The `atomicTransactions` API also supports interactions with smart contracts that are deployed on Ethereum networks. You can use this functionality to transfer non-fungible tokens (NFTs) that were minted in Hyperledger Fabric chaincode on Oracle Blockchain Platform to an Ethereum or Polygon network, by invoking two transactions atomically. NFT attributes such as the token ID, price, and token history can also be passed from Oracle Blockchain Platform to Ethereum atomically. After you transfer an NFT from Oracle Blockchain Platform to Ethereum, the NFT can be listed on a public NFT marketplace.

To transfer an NFT from Oracle Blockchain Platform to Ethereum, you use two basic steps in one atomic transaction:

1. Burn the NFT on Oracle Blockchain Platform. Call the `burnNFT` method, to burn (delete) the NFT from the Hyperledger Fabric chaincode on Oracle Blockchain Platform. Oracle Blockchain Platform supports NFTs in enhanced versions of two standards, ERC-721 and ERC-1155, with the Blockchain App Builder tool. For more information on the burnNFT method, see the relevant topic in *Blockchain App Builder for Oracle Blockchain Platform*:
   - Scaffolded TypeScript NFT Project for ERC-721
   - Scaffolded Go NFT Project for ERC-721
   - Scaffolded TypeScript NFT Project for ERC-1155
   - Scaffolded Go NFT Project for ERC-1155
2. Mint the NFT on Ethereum. Call a smart contract on the Ethereum or Polygon network to mint the NFT on that network, using the parameters returned by the `burnNFT` method. Sample versions of smart contracts written in the Solidity language for NFTs are available in the following archive file: solidity-smartcontracts-fab253.zip. The smart contracts, one for each of the enhanced token standards ERC-721 and ERC-1155, include a `mintNFT` method, which creates NFTs with custom properties such as price and token history, which can be fetched from the output of the `burnNFT` method in the previous step.
   For unsigned requests, if the custom properties are in the `ParamKeys` parameter and corresponding dynamic parameters are passed in the `params` parameter, the atomic transactions API can fetch the parameters from the `burnNFT` method and send them to the Ethereum smart contract. The `mintNFT` method takes the following arguments:
   - `to` – The Ethereum address for the account where the NFT will be minted.

- `id` – The token ID of the NFT.
- `price` – The price of the NFT.
- `tokenHistory` – The history of the NFT from the Oracle Blockchain Platform chaincode.

The smart contract requires that the token ID of the NFT must be a numeric string (a string that can be converted to an integer). For example a token ID can be `2` but not `token2`.

The token URI of the NFT in the chaincode deployed on Oracle Blockchain Platform must follow a certain format to make it compatible with Solidity smart contracts:

- ERC-1155: A URI for all token types that relies on ID substitution, such as `https://token-cdn-domain/{id}.json`.
- ERC-721: A URI where all tokens share a prefix (a base URI) followed by a token URI, such as `http://api.myproject.example.com/token/<tokenURI>`.

You can use the Remix IDE to generate an application binary interface (ABI) for the smart contract. The ABI can then be used with the `atomicTransactions` API. If you change any method in the smart contract, you must recompile the contract and generate the ABI again.

For more information about the parameters to use for Ethereum transactions in an atomic workflow, including an example of transferring an NFT to an Ethereum network, see Atomic Transactions REST Endpoints.

Include Oracle Blockchain Platform in Global Distributed Transactions

Your application might need to make updates across the Oracle Blockchain Platform ledger and other repositories such as databases or other blockchain ledgers in an atomic fashion, where either all updates succeed or none do.

To enable atomic updates across multiple databases, developers use global transactions that are coordinated by distributed transaction coordinators such as Oracle WebLogic Server, Oracle Tuxedo, Oracle Transaction Manager for Microservices, JBoss Enterprise Application Platform, IBM WebSphere, and other systems. All of these systems rely on the X/Open XA protocol to orchestrate a two-phase commit process by using standard APIs that are provided by XA Resource Managers (RMs) for each database or other resource. Oracle Blockchain Platform supports two-phase commits and provides its own XA RM library, which external transaction coordinators can use to invoke XA-compliant APIs. These global transactions can also include a single non-XA resource (for example, a non-Oracle blockchain ledger or non-XA compliant database) by using a last resource commit optimization.

The XA specification is part of the X/Open Distributed Transaction Processing architecture, which defines a standard architecture that enables multiple application programs to share resources provided by multiple resource managers. The Java XA interface itself is defined as part of the Java platform. For more information on the Java XA interface, see Interface XAResource in the Java documentation.

Oracle Blockchain Platform provides a library that conforms to the XA specification and implements the standard Java interface for an XA resource manager. The library enables a client-side transaction manager to coordinate global transactions. A global transaction is a single unit of work that might include operations such as database updates and blockchain transactions, all of which must be committed atomically. In other words, all of the operations must succeed to be committed. If any operation that is part of the global transaction fails, then all operations are rolled back. The XA interface relies on the two-phase commit protocol, similar to the protocol supported by the atomic transactions REST endpoints. For more

information about atomic transactions in Oracle Blockchain Platform, see Make Atomic Updates Across Chaincodes and Channels.

The XA implementation for Oracle Blockchain Platform is supplied as a Java library, downloadable from the **Developer Tools** tab on the **Application Development** pane of the Oracle Blockchain Platform console.

Full details on the library are included in the Javadoc information supplied in the downloadable file. The three key objects supported by the library are `OBPXAResource`, `OBPXADataSource`, and `OBPXAConnection`.

Object	Purpose
`OBPXAResource`	This class implements the required APIs for a transaction manager to coordinate with Oracle Blockchain Platform as a resource manager for XA transactions.
`OBPXADataSource`	Use this object to get an instance of the `OBPXAConnection` and to specify the authentication and authorization credentials.
`OBPXAConnection`	Use this object to get an instance of the `OBPXAResource` object and to define the blockchain transactions to run as part of an XA transaction.

To use the XA library with Oracle Blockchain Platform, the application must provide credentials for authentication and authorization of the requested operations. The library supports both basic authentication (user/password) and OAuth 2.0 access tokens, which you can configure when you create the `OBPXADataSource` instance. The two authentication methods are consistent with the authentication methods that you use with the Oracle Blockchain Platform REST proxy. You can use basic authentication for testing and internal development purposes. Do not use basic authentication in production environments. For more information, see Authentication in the REST API documentation.

After you create an `OBPXADataSource` instance, you can use the `obpxaDataSource.getXAConnection()` method to get the `xaConnection` instance. To update authentication when using OAuth 2.0 access tokens, you can use the `getXAConnection` method, as shown in the following code:

```
OBPXAConnection xaConnection =
obpxaDataSource.getXAConnection(accessToken);      // get an XA connection
using an OAuth 2.0 access token
```

You can also use the `getXAConnection` method to update basic authentication.

```
OBPXAConnection xaConnection = obpxaDataSource.getXAConnection(user,
password);   // get an XA connection using username and password for basic
authentication
```

To define a blockchain transaction to be run as part of a global XA transaction, you use the following method:

```
public void createXAInvokeTransaction(Xid xid, OBPXAInvokeTxRequest
invokeTxRequest)
```

In this method, `xid` is a global transaction identifier and `invokeTxRequest` is the blockchain transaction to be run as part of the global XA transaction. To create an XA invoke transaction request, you use the following constructor method:

```
OBPXAInvokeTxRequest invokeTxRequest = new OBPXAInvokeTxRequest(channel,
chaincode, args);
```

In this constructor method, `channel` is the channel where the blockchain transaction will run, `chaincode` is the chaincode to use, and `args` includes the chaincode function and any arguments to use for the transaction.

The following snippet of code demonstrates creating an `OBPXADataSource` object, getting the `OBPXAConnection` instance, and then creating the transaction request and calling the `createXAInvokeTransaction` method.

```
OBPXADataSource obpxaDataSource = OBPXADataSource.builder()
                        .withHost(host)
                        .withPort(port)
                        .withBasicAuth(username, password)
                        .withRole(role)
                        .build();
.
.
.
OBPXAConnection obpxaConnection = obpxaDataSource.getXAConnection();

OBPXAInvokeTxRequest invokeTxRequest = new OBPXAInvokeTxRequest(channel,
chaincode, args);
invokeTxRequest.setEndorsers(endorsersArray); // optional blockchain
transaction request attributes
invokeTxRequest.setTransientMap(transientMap); // optional blockchain
transaction request attributes
invokeTxRequest.setTimeout(60000); // optional blockchain transaction request
attributes

obpxaConnection.createXAInvokeTransaction(xid, invokeTxRequest);
```

Scenario: Explore XA Transactions Using Samples

The following scenario is similar to the one described for atomic transactions: Scenario: Explore Atomic Transactions Using Samples, which uses the Balance Transfer and Marbles samples that are included with Oracle Blockchain Platform.

In this scenario, you install the Balance Transfer and Marbles samples on two different instances of Oracle Blockchain Platform. Each instance then corresponds to an XA data source:

- XA resource OBP-1, with the Marbles chaincode installed on the `goods` channel
- XA resource OBP-2, with the Balance Transfer chaincode installed on the `wallet` channel

In this scenario, you can use an XA transaction that spans multiple data sources to ensure that the exchange of funds and the marble transfer occur in an atomic manner, where either all operations succeed or none succeed. The following code illustrates this scenario:

```
OBPXADataSource obpxaDS1 = ... // create an XA data source, supplying details
about the OBP-1 instance
```

```
OBPXADataSource obpxaDS2 = ... // create an XA data source, supplying details
about the OBP-2 instance

// start a global transaction in the client application
// invoke marble transfer on OBP-1
OBPXAConnection obpxaConn1 = (OBPXAConnection) obpxaDS1.getXAConnection();
OBPXAInvokeTxRequest invokeMarbleTransferReq = new
OBPXAInvokeTxRequest("goods", "obcs-marbles", new String[]{"transferMarble",
"marble1", "tom"});
obpxaConn1.createXAInvokeTransaction(xid1, invokeMarbleTransferReq);
.
.
.
// invoke fund transfer on OBP-2
OBPXAConnection obpxaConn2 = (OBPXAConnection) obpxaDS2.getXAConnection();
OBPXAInvokeTxRequest invokeBalanceTransferReq = new
OBPXAInvokeTxRequest("wallet", "obcs-example02", new String[]{"invoke", "a",
"b", "50"});
obpxaConn2.createXAInvokeTransaction(xid2, invokeBalanceTransferReq);
.
.
.
// end the global transaction in the client application
```

9

Work With Databases

This topic contains information to help you understand how to query the state database and how to create and configure a rich history database.

Topics:

Query the State Database

This topic contains information to help you understand how to query the state database where the blockchain ledger's current state data is stored.

What's the State Database?

The blockchain ledger's current state data is stored in the state database.

When you develop Oracle Blockchain Platform chaincodes, you can extract data from the state database by executing rich queries. Oracle Blockchain Platform supports rich queries by using the SQL rich query syntax and the CouchDB find expressions. See SQL Rich Query Syntax and CouchDB Rich Query Syntax.

Hyperledger Fabric doesn't support SQL rich queries. If your Oracle Blockchain Platform network contains Hyperledger Fabric participants, then you need to make sure to do the following:

- If your chaincodes contain SQL rich query syntax, then those chaincodes are installed only on member peers using Oracle Blockchain Platform.
- If a chaincode needs to be installed on Oracle Blockchain Platform and Hyperledger Fabric peers, then use CouchDB syntax in the chaincodes and confirm that the Hyperledger Fabric peers are set up to use CouchDB as their state database repository. Oracle Blockchain Platform can process CouchDB.

How Does Oracle Blockchain Platform Work with Berkeley DB?

Oracle Blockchain Platform uses Oracle Berkeley DB as the state database. Oracle Blockchain Platform creates relational tables in Berkeley DB based on the SQLite extension. This architecture provides a robust and performant way to validate SQL rich queries.

For each channel chaincode, Oracle Blockchain Platform creates a Berkeley DB table. This table stores state information data, and contains at least a key column named `key`, and a value column named `value` or `valueJson`, depending on whether you're using JSON format data.

Column Name	Type	Description
`key`	TEXT	Key column of the state table.
`value`	TEXT	Value column of the state table.

Column Name	Type	Description
`valueJson`	TEXT	JSON format value column of the state table.

Note that the `valueJson` and `value` columns are mutually-exclusive. So, if the chaincode assigns a JSON value to a key, then the `valueJson` column will hold that value, and the value column will be set to null. If the chaincode assigns a non-JSON value to a key, then the `valueJson` column will be set to null, and the value column will hold the value.

Example of a State Database

These are examples of keys and their values from the Car Dealer sample's state database:

key	value	valueJson
abg1234	null	{"docType": "vehiclePart", "serialNumber": "abg1234", "assembler": "panama-parts", "assemblyDate": 1502688979, "name": "airbag 2020", "owner": "Detroit Auto", "recall": false, "recallDate": 1502688979}
abg1235	null	{"docType": "vehiclePart", "serialNumber": "abg1235", "assembler": "panama-parts", "assemblyDate": 1502688979, "name": "airbag 4050", "owner": "Detroit Auto", "recall": false, "recallDate": 1502688979}
ser1236	null	{"docType": "vehiclePart", "serialNumber": "ser1236", "assembler": "panama-parts", "assemblyDate": 1502688979, "name": "seatbelt 10020", "owner": "Detroit Auto", "recall": false, "recallDate": 1502688979}
bra1238	null	{"docType": "vehiclePart", "serialNumber": "bra1238", "assembler": "bobs-bits", "assemblyDate": 1502688979, "name": "brakepad 4200", "owner": "Detroit Auto", "recall": false, "recallDate": 1502688979}
dtrt10001	null	{"docType": "vehicle", "chassisNumber": "dtrt10001", "manufacturer": "Detroit Auto", "model": "a coupe", "assemblyDate": 1502688979, "airbagSerialNumber": "abg1235", "owner": "Sam Dealer", "recall": false, "recallDate": 1502688979

Rich Queries in the Console

Administrators can run and analyze rich queries from the console.

1. Go to the console and select the **Channels** tab.
2. In the channels table, locate the channel where you want to run a query, click the channels **More Actions** button, and then click **Analyze Rich Queries**. The Analyze Rich Queries dialog box is displayed.
3. To run a rich query against the state database, select **Query Execution**.
   a. For **Chaincode**, select the chaincode that is deployed to the channel that you want to query.
   b. For **Peer**, select the peer to query.

Only peers in the current organization that are running the selected chaincode are available.

c. For **Rich Query**, enter the rich query to run and analyze.

The query format must follow the rich query syntax. For more information about rich query syntax, see Supported Rich Query Syntax.

d. For **Result Rows Limit**, move the slider to the maximum number of result rows to fetch. You can fetch up to 50 rows of results.

4. To get the execution plan for a query, select **Query Plan Explain**. A query execution plan is the sequence of operations that was performed to run the query.

a. For **Chaincode**, select the chaincode that is deployed to the channel that you want to query.

b. For **Peer**, select the peer to query.

c. For **Collection**, select the state database or private data collection.

d. For **Rich Query**, enter the rich query.

The `explain` keyword is not needed for this query.

For example: `select * from <state>`

5. Click **Execute**. The **Results** field shows the query result table or the execution plan. To export the results table as a `.csv` file, click **Export**.

The results table size is limited to 1 MB. You might need to refine your query to avoid exceeding this limit.

Supported Rich Query Syntax

Oracle Blockchain Platform supports two types of rich query syntax that you can use to query the state database: SQL rich query and CouchDB rich query.

SQL Rich Query Syntax

The Berkeley DB JSON extensions are in the form of SQL functions.

Before You Begin

Note the following information:

- You can only access the channel chaincode (<STATE>) that you're executing your query from.
- Only the SELECT statement is supported.
- You can't modify the state database table.
- A rich query expression can have only one SELECT statement.
- The examples in this topic are just a few ways that you can write your rich query. You've access to the usual full SQL syntax to query a SQL database.
- You've access to the JSON1 Extension (SQLite extension). See JSON1 Extension and SQL As Understood by SQLite.

If you need more information about writing and testing chaincodes, see Develop Chaincodes.

How to Refer to the State Database in Queries

The state database table name is internally managed by Oracle Blockchain Platform, so you don't need to know the state database's physical name when you write a chaincode.

Instead, you must use the `<STATE>` alias to refer to the table name. For example: `select key, value from <STATE>`.

Note that the `<STATE>` alias is **not** case-sensitive, so you can use either `<state>`, `<STATE>`, or something like `<StAtE>`.

Retrieve All Keys

Use this syntax:

```
SELECT key FROM <STATE>
```

For example, if you use this syntax to query the Car Dealer sample, then you'll get the following list of keys:

key

abg1234

abg1235

ser1236

bra1238

dtrt10001

Retrieve All Keys and Values Ordered Alphabetically by Key

Use this syntax:

```
SELECT key AS serialNumber, valueJson AS details FROM  <state> ORDER BY key
```

For example, if you use this syntax to query the Car Dealer sample, then you'll get the following results:

serialNumber	details
abg1234	{"docType": "vehiclePart", "serialNumber": "abg1234", "assembler": "panama-parts", "assemblyDate": 1502688979, "name": "airbag 2020", "owner": "Detroit Auto", "recall": false, "recallDate": 1502688979}
abg1235	{"docType": "vehiclePart", "serialNumber": "abg1235", "assembler": "panama-parts", "assemblyDate": 1502688979, "name": "airbag 4050", "owner": "Detroit Auto", "recall": false, "recallDate": 1502688979}
bra1238	{"docType": "vehiclePart", "serialNumber": "bra1238", "assembler": "bobs-bits", "assemblyDate": 1502688979, "name": "brakepad 4200", "owner": "Detroit Auto", "recall": false, "recallDate": 1502688979}
dtrt10001	{"docType": "vehicle", "chassisNumber": "dtrt10001", "manufacturer": "Detroit Auto", "model": "a coupe", "assemblyDate": 1502688979, "airbagSerialNumber": "abg1235", "owner": "Sam Dealer", "recall": false, "recallDate": 1502688979
ser1236	{"docType": "vehiclePart", "serialNumber": "ser1236", "assembler": "panama-parts", "assemblyDate": 1502688979, "name": "seatbelt 10020", "owner": "Detroit Auto", "recall": false, "recallDate": 1502688979}

Retrieve All Keys and Values Starting with "abg"

Use this syntax:

```
SELECT key AS serialNumber, valueJson AS details FROM <state> WHERE key LIKE
'abg%'SELECT key, value FROM <STATE>
```

For example, if you use this syntax to query the Car Dealer sample, then you'll get the following results:

serialNumber	details
abg1234	{"docType": "vehiclePart", "serialNumber": "abg1234", "assembler": "panama-parts", "assemblyDate": "1502688979", "name": "airbag 2020", "owner": "Detroit Auto", "recall": "false", "recallDate": "1502688979"}
abg1235	{"docType": "vehiclePart", "serialNumber": "abg1235", "assembler": "panama-parts", "assemblyDate": "1502688979", "name": "airbag 4050", "owner": "Detroit Auto", "recall": "false", "recallDate": "1502688979"}

Retrieve All Keys with Values Containing a Vehicle Part Owned by "Detroit Auto"

Use this syntax:

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

For example, if you use this syntax to query the Car Dealer sample, then you'll get the following list of keys:

key

abg1234

abg1235

ser1236

bra1238

Retrieve Model and Manufacturer for all Cars Owned by "Sam Dealer"

Use this syntax:

```
SELECT json_extract(valueJson, '$.model') AS model, json_extract(valueJson,
'$.manufacturer') AS manufacturer FROM <state> WHERE json_extract(valueJson,
'$.docType') = 'vehicle' AND json_extract(valueJson, '$.owner') = 'Sam Dealer'
```

For example, if you use this syntax to query the Car Dealer sample, then you'll get the following results:

model	manufacturer
a coupe	Detroit Auto

If the state value is JSON array, you may use this syntax to retrieve model and manufacturer for all cars owned by "Sam Dealer":

```
SELECT json_extract(j.value, '$.model') AS model, json_extract(j.value,
'$.manufacturer') AS manufacturer FROM <state> s,
json_each(json_extract(s.valueJson,'$')) j WHERE json_valid(j.value) AND
json_extract(j.value, '$.owner') = 'Sam Dealer'
```

CouchDB Rich Query Syntax

Use the information in this topic if you're migrating your chaincodes containing CouchDB syntax to Oracle Blockchain Platform, or if you need to write chaincodes to install on Hyperledger Fabric peers participating in an Oracle Blockchain Platform network.

If you're writing a new chaincode, then Oracle recommends that you use SQL rich queries to take advantage of the performance benefits that Oracle Blockchain Platform with Berkeley DB provides.

If you need more information about writing and testing chaincodes, see Develop Chaincodes.

Unsupported Query Parameters and Selector Syntax

Oracle Blockchain Platform doesn't support the `use_index` parameter. If used, Oracle Blockchain Platform ignores this parameter, and it will automatically pick the indexes defined on the StateDB in question.

Parameter	Type	Description
use_index	json	Instructs a query to use a specific index.

Retrieve All Models, Manufacturers, and Owners of Cars, and Order Them by Owner

Use this expression:

```
{
  "fields": ["model", "manufacturer", "owner"],
  "sort": [
    "owner"
   ]
}
```

Retrieve Model and Manufacturer for All Cars Owned by "Sam Dealer"

Use this expression:

```
{
  "fields": ["model", "manufacturer"],
  "selector": {
    "docType"  : "vehicle",
     "owner" : "Sam Dealer"
  }
}
```

State Database Indexes

The state database can contain a large amount of data. In such cases Oracle Blockchain Platform uses indexes to improve data access.

Default Indexes

When a chaincode is deployed, Oracle Blockchain Platform creates two indexes.

- Key index: Created on the key column.

- Value index: Created on the value column.

Custom Indexes

In some cases, you might need to create custom indexes. You define these indexes by using any expression that can be resolved in the context of the state table. Custom indexes created against Berkeley DB rely on the SQLite syntax, but they otherwise follow the same CouchDB implementation provided by Hyperledger Fabric.

You can use custom indexes to dramatically improve the performance of WHERE and ORDER BY statements on large data sets. Because using custom indexes slows down data insertions, use them judiciously.

Each custom index is defined as an array of expressions, which support compound indexes, expressed as a JSON document inside one file. There is one index per file. You must package this file with the chaincode in a folder named `indexes` in the following directory structure: `statedb/relationaldb/indexes`. For more information, see How to add CouchDB indexes during chaincode installation.

Example Custom Indexes

The custom index examples in this section use the Car Dealer sample.

Example 1: This example indexes the use of the `json_extract` expression in the context of WHERE and ORDER BY expressions.

```
{"indexExpressions": ["json_extract(valueJson, '$.owner')"]}
```

For example:

```
SELECT … FROM … ORDER BY json_extract(valueJson, '$.owner')
```

Example 2: This example indexes the compound use of the two `json_extract` expressions in the context of WHERE and ORDER BY expressions.

```
{"indexExpressions": ["json_extract(valueJson, '$.docType')",
"json_extract(valueJson, '$.owner')"]}
```

For example:

```
SELECT … FROM … WHERE json_extract(valueJson, '$.docType') = 'vehiclePart' AND
json_extract(valueJson, '$.owner') = 'Detroit Auto'
```

Example 3: This example creates both the index described in Example 1 and the index described in Example 2. Each JSON structure must be included in a separate file. Each file describes a single index: a simple index like Example 1, or a compound index like Example 2.

Index 1: `{"indexExpressions": ["json_extract(valueJson, '$.owner')"]}`

Index 2: `{"indexExpressions": ["json_extract(valueJson, '$owner')",`
`"json_extract(valueJson, '$.docType')"]}`

In the following example, Index 2 is applied to the `AND` expression in the `WHERE` portion of the query, while Index 1 is applied to the `ORDER BY` expression:

```
SELECT … FROM … WHERE json_extract(valueJson, '$.docType') = 'vehiclePart' AND
json_extract(valueJson, '$.owner') = 'Detroit Auto' ORDER BY
json_extract(valueJson, '$.owner')
```

JSON Document Format

The JSON document must be in the following format:

```
{"indexExpressions": [expr1, ..., exprN]}
```

For example:

```
{"indexExpressions": ["json_extract(valueJson, '$.owner')"]}
```

Differences in the Validation of Rich Queries

In some cases, the standard Hyperledger Fabric with CouchDB rich query and the Oracle Berkeley DB rich query behave differently.

In standard Hyperledger Fabric with CouchDB, each key and value pair returned by the query is added to the transaction's read-set and is validated at validation time and without re-executing the query. In Berkeley DB, the returned key and value pair isn't added to the read-set, but the rich query's result is hashed in a Merkle tree and validated against the re-execution of the query at validation time.

Native Hyperledger Fabric doesn't provide data protection for rich query. However, Berkeley DB contains functionality that protects and validates the rich query by adding the Merkle tree hash value into the read-set, re-executing the rich query, and at the validation stage re-calculating the Merkle tree value. Note that because validation is more accurate in Oracle Blockchain Platform with Berkeley DB, chaincode invocations are sometimes flagged for more frequent phantom reads.

Create the Fallback State Database

You can connect to Oracle Database to create a fallback state database. The hybrid state database model uses a fallback state database, which can become the primary state database if there are any issues with the embedded state database on the peer.

What's the Fallback State Database?

The fallback state database maintains a secondary copy of the state database in Oracle Database, while the primary state database is stored on the embedded Berkeley DB.

The state database is stored on each peer for all channels that the peer is joined to. Oracle Blockchain Platform uses Berkeley DB as the embedded database on peer nodes. If a peer crashes or restarts, the state database can get corrupted. Oracle Blockchain Platform automatically detects and rebuilds a corrupted state database from the ledger, but this can take a significant amount of time depending on the ledger size and number of blocks. The peer node is not available for endorsing or committing transactions during the rebuild process.

The hybrid state database model adds an external Oracle Database as a fallback. In normal operation, peers complete synchronous block commits to the Berkeley DB state database and asynchronous commits to the fallback database. If the embedded state database fails, the peer automatically switches to use Oracle Database for synchronous commits while the Berkeley DB state database is asynchronously rebuilt. After the rebuild process completes, the peer switches back to normal operation.

You must use Oracle Autonomous AI Transaction Processing as the fallback database.

Enable the Fallback State Database

Use the console to provide database connection information and select the peers where you want to configure a fallback state database.

1. Go to the console and click the **More Actions** icon in the title bar, where the name of the instance is also displayed.
2. Click **Configure Fallback State Database**.

   The Configure Fallback State Database window is displayed.
3. Specify the connection information for Oracle Autonomous Transaction Processing.
   a. If you have already configured the rich history database and you want to use the same connection information for the fallback state database, click **Use Rich History Database Configuration** to use the same instance of Oracle Database as the fallback state database. Otherwise, specify a **User Name**, **Password**, and **Connection String**, and optionally upload a wallet file. For more information on connection strings, see Create the Oracle Base Database Service Connection String.
   b. To configure the fallback database for every peer in the network, click **Enable for all peers**.
   c. To configure the fallback database for any newly added (scaled) peer, click **Enable for newly scaled peers**.
   d. Click **Save**.

      All selected peers restart when you apply the configuration.
4. To enable or disable the fallback state database on a specific peer, edit the peer configuration.
   a. Click the **Nodes** tab.
   b. In the nodes table, for the peer that you want to modify click the **More Actions** icon and then click **Edit Configuration**.
   c. Under **Fallback State Database**, select `ENABLE` or `DISABLE`, and then click **Submit**.

Monitor the State Database

After you configure a fallback state database on a peer node, you can monitor the state database status.

You must configure a fallback state database to monitor the state database status.

1. Go to the console and click the **Nodes** tab.
2. In the nodes table, click the **More Actions** icon for the peer node that you want to monitor, and then click **Monitor State Database**.

   A table is displayed that contains the following information about the state database status.

   Channel Name
   The channel that the peer is joined to.

   Active Database
   The database that is currently accepting synchronous block commits, either the primary database (Berkeley DB) or the fallback database (Oracle Database).

   Ledger Block Height
   The number of blocks currently stored in the ledger.

   Primary Database State

   - `SYNC_COMMITS`: The database is operating normally in synchronous mode.

- `ASYNC_RECOVERY`: The database is processing commits in asynchronous mode and attempting to catch up to the ledger block height.
- `ASYNC_ABORTED`: A persistent error occurred while the database was processing commits in asynchronous mode, or the asynchronous block queue is full. In either case, the underlying issue must be corrected and the peer must be restarted.

Primary Database Block Height
The number of blocks currently stored in the primary state database.

Fallback Database State

- `ASYNC_COMMITS`: The database is operating normally as a fallback, in asynchronous mode.
- `SYNC_COMMITS`: The database is operating in synchronous mode, because there was a problem with the primary database or because the primary database block height was below the ledger block height while the fallback database ledger height was equal to the ledger block height.
- `ASYNC_RECOVERY`: The database is processing commits in asynchronous mode and attempting to catch up to the ledger block height.
- `ASYNC_ABORTED`: A persistent error occurred while the database was processing commits in asynchronous mode, or the asynchronous block queue is full. In either case, the underlying issue must be corrected and the peer must be restarted.

Fallback Database Block Height
The number of blocks currently stored in Oracle Database (the fallback database).

Async Queue Length
The number of blocks in the queue awaiting asynchronous processing by the current asynchronous database.

Last Async Error
The most recent error related to the asynchronous database, which can include connection or credentials issues or problems with the asynchronous queue or key sizes.

Create the Rich History Database

This topic contains information to help you specify an Oracle Database connection and choose channels to create the rich history database. You'll use this database to make analytics reports and visualizations of your ledger's activities.

What's the Rich History Database?

The rich history database is external to Oracle Blockchain Platform and contains data about the blockchain ledger's transactions on the channels you select. You use this database to create analytics reports and visualization about your ledger's activities.

For example, using the rich history database, you could create analytics to learn the average balance of all of the customers in your bank over some time interval, or how long it took to ship merchandise from a wholesaler to a retailer.

Internally, Oracle Blockchain Platform uses the Hyperledger Fabric history database to manage the ledger and present ledger transaction information to you in the console. Only the chaincodes can access this history database, and you can't expose the Hyperledger Fabric history database as a data source for analytical queries. The rich history database uses an external copy of Oracle Database and contains many details about every transaction

committed on a channel. This level of data collection makes the rich history database an excellent data source for analytics. For information about the data that the rich history database collects, see Rich History Database Tables and Columns.

You can only use a database such as Oracle Autonomous AI Lakehouse or Oracle Base Database Service with Oracle Cloud Infrastructure to create your rich history database. You use the Oracle Blockchain Platform console to provide the connection string and credentials to access and write to the database. Note that the credentials you provide are the database's credentials and Oracle Blockchain Platform doesn't manage them. After you create the connection, you'll select the channels that contain the ledger data that you want to include in the rich history database. See Enable and Configure the Rich History Database.

You can use standard tables or blockchain tables to store the rich history database. Blockchain tables are tamperproof append-only tables, which can be used as a secure ledger while also being available for transactions and queries with other tables.

You can use any analytics tool, such as Oracle Analytics Cloud or Oracle Data Visualization Cloud Service, to access the rich history database and create analytics reports or data visualizations.

Create the Oracle Base Database Service Connection String

You must collect information from the Oracle Base Database Service deployed on Oracle Cloud Infrastructure to build the connection string required by the rich history database. You must also enable access to the database through port 1521.

Find and Record Oracle Base Database Service Information

The information you need to create a connection to the Oracle Base Database Service is available in the Oracle Cloud Infrastructure Console.

1. From the Infrastructure Console, select **Database** from the navigation menu.
2. Locate the database that you want to connect to and record the **Public IP** address.
3. Select the name of the database that you want to connect to and record the values in these fields:
   - **Database Unique Name**
   - **Host Domain Name**
   - **Port**
4. Find a user name and password of a database user (for example, user SYSTEM) with permissions to read from this database, and make a note of these.

Enable Database Access Through Port 1521

Add an ingress rule that enables the rich history database to access the database through port 1521.

1. In the Oracle Cloud Infrastructure home page, select the navigation icon and then under **Databases** select **DB Systems**.
2. Select the database that you want to connect to.
3. Select the **Virtual Cloud Network** link.
4. Navigate to the appropriate subnet, and then under **Security Lists**, select **Default Security List For <Target Database>**.
   The Security List page is displayed.

5. Select **Edit All Rules**.
6. Add an ingress rule to allow any incoming traffic from the public internet to reach port 1521 on this database node, with the following settings:
   - **SOURCE CIDR**: 0.0.0.0/0
   - **IP PROTOCOL**: TCP
   - **SOURCE PORT RANGE**: All
   - **DESTINATION PORT RANGE**: 1521
   - **Allows**: TCP traffic for ports: 1521

Build the Connection String

After enabling access to Oracle Database, use the information that you collected previously to build the connection string in the Configure Rich History dialog box.

Construct the connection string using the following syntax: *<publicIP>:<portNumber>/<database unique name>.<host domain name>*

The following connection string is an example: `192.0.2.0:1521/CustDB_iad1vm.sub05031027070.customervcnwith.oraclevcn.example.com`

Check Database User Privileges

In order for the rich history functionality to be able to manage its database sessions and to recover from temporary database or network downtime, ensure that the database user registered with Oracle Blockchain Platform has the following two privileges:

```
grant select on v_$session to <user>;
grant alter system to <user>;
```

Additionally, if the rich history database uses Oracle Autonomous AI Lakehouse, the database user must have the following privilege:

```
grant unlimited tablespace to <user>;
```

If the database user doesn't have those privileges already, they must be granted by the system database administrator.

Without these privileges, Oracle Blockchain Platform can replicate to the database but it cannot recover from situations leading to a damaged database session, which prevents the rich history from catching up with recent transactions for an extended period. Without these privileges on Oracle Autonomous AI Lakehouse, no rich history data is saved.

Enable and Configure the Rich History Database

Use the console to provide database connection information and select the channels with the chaincode ledger data that you want to write to the rich history database. By default, channels aren’t enabled to write data to the rich history database.

- Each blockchain network member configures its own rich history database.
- You must use Oracle Database. No other database types are supported.
- Each channel that writes to the rich history database must contain at least one peer node.

1. Enter connection and credential information for the instance of Oracle Database to use to store rich history information.
   a. Go to the console and select **Options**, and then select **Configure Rich History**.

      The Configure Rich History dialog box is displayed.
   b. Enter the user name and password required to access Oracle Database.
   c. In the **Connection String** field, enter the connection string for the database that you'll use to store rich history data. What you enter here depends on the database you're using.
      - If you're using Oracle Autonomous AI Lakehouse, then you'll enter something similar to <*username*>adw_high. To find Oracle Autonomous AI Lakehouse's connection information, go to its credential wallet ZIP file and open the TNS file.
      - If you're using Oracle Base Database Service with Oracle Cloud Infrastructure (OCI), see Create the Oracle Base Database Service Connection String.
      - If you're using a non-autonomous database (a database that doesn't use a credential wallet) and want to use the `sys` user to connect to the database, then you must append `?as=sys[dba|asm|oper]` to the connection string. For example, `203.0.113.0:1521/oraclevcn.example.com?as=sysdba`
   d. If you're using an OCI autonomous database instance (for example, Oracle Autonomous AI Lakehouse or Oracle Autonomous AI Transaction Processing), then use the **Wallet Package File** field to upload the required credential wallet `.zip` file. This file contains client credentials and is generated by the database.

      > **Note**
      >
      > When you open the Configure Rich History dialog box again after you configure rich history, the wallet file name is not displayed. If you update other settings, you must upload the wallet `.zip` file again before selecting **Save**. If you select **Save** while no wallet file name is displayed, the configuration is updated not to use a wallet file.

   e. To use blockchain tables to store the rich history database, select **Use Database Blockchain Table**.

      The underlying database must support blockchain tables.
      - To specify the number of days to retain tables and rows, select **Basic Configuration**, and then enter the number of days to retain tables and rows. Enter `0` to retain tables or rows permanently. To prevent further changes to the retention values, select **Locked**.
      - To specify table and row retention by using a data definition language (DDL) statement, select **Advanced Configuration Query** and then enter the DDL statement.
   f. Select **Save**.
2. Enable rich history on the channels that contain the chaincode data that you want to write to the rich history database.
   a. Go to the console and select the **Channels** tab.
   b. Locate the channel that contains the chaincode data that you want to write to the rich history database. Select **More Options** and then select **Configure Rich History**.

      The Configure Rich History dialog is displayed.

c. Select **Enable Rich History**. To store private data collections in the rich history database, enter a list of private data collection names, separated by commas. For more information about private data collections, see What Are Private Data Collections?. To add transaction details to the rich history database, select the details that you want added. Select **Save**.

The rich history database is configured, but tables are not created in the database immediately. When the next relevant transaction or ledger change happens, the tables are created in the rich history database.

Modify the Connection to the Rich History Database

You can change the rich history database's connection information.

After tables are created in the database for a channel, modifying the rich history configuration for the channel has no effect, even after you select **Save**, unless you change the user name and password or the connection string. If you change the user name and password, tables are created in the same database. If you change the connection string and credentials, a different database is configured, and tables are created after the next relevant transaction or ledger change. You cannot change a rich history database from standard tables to blockchain tables, and you cannot change retention times, unless you also change the credentials or connection string.

1. Go to the console and select **Options**, and then select **Configure Rich History**.
2. Update the user name and password as needed to access the database.
3. If needed, in the **Connection String** field, modify the connection string for the database that you'll use to store rich history data. What you enter here depends on the database you're using.
   - If you're using Oracle Autonomous AI Lakehouse, then you'll enter something similar to <*username*>adw_high. To find Oracle Autonomous AI Lakehouse's connection information, go to its credential wallet `.zip` file and open the TNS file.
   - If you're using Oracle Base Database Service with Oracle Cloud Infrastructure (OCI), see Create the Oracle Base Database Service Connection String.
   - If you're using a non-autonomous database (a database that doesn't use a credential wallet) and want to use the `sys` user to connect to the database, then you must append `?as=sys[dba|asm|oper]` to the connection string. For example, `203.0.113.0:1521/oraclevcn.example.com?as=sysdba`
4. If you're using an OCI autonomous database instance (for example, Oracle Autonomous AI Lakehouse or Oracle Autonomous AI Transaction Processing), then use the **Wallet Package File** field to upload or re-upload the required credential wallet file. This file contains client credentials and is generated by the database.

   > **Note**
   >
   > When you open the Configure Rich History dialog box again after you configure rich history, the wallet file name is not displayed. If you update other settings, you must upload the wallet `.zip` file again before selecting **Save**. If you select **Save** while no wallet file name is displayed, the configuration is updated not to use a wallet file.

5. To use blockchain tables to store the rich history database, select **Use Database Blockchain Table**.

The underlying database must support blockchain tables.

- To specify the number of days to retain tables and rows, select **Basic Configuration**, and then enter the number of days to retain tables and rows. Enter `0` to retain tables or rows permanently. To prevent further changes to the retention values, select **Locked**.
- To specify table and row retention by using a data definition language (DDL) statement, select **Advanced Configuration Query** and then enter the DDL statement.

6. Select **Save**.

Configure the Channels that Write Data to the Rich History Database

You can enable channels to write chaincode ledger data to the rich history database, and you can stop channels from writing data to the rich history database. You can also configure an individual channel to use a different rich history database configuration than the global setting.

You must specify the global information to connect to the rich history database before you can select channels that write to the rich history database. See Enable and Configure the Rich History Database.
After tables are created in the database for a channel, modifying the rich history configuration for the channel has no effect, even after you click **Save**, unless you change the user name and password or the connection string. If you change the user name and password, tables are created in the same database. If you change the connection string and credentials, a different database is configured, and tables are created after the next relevant transaction or ledger change. You cannot change a rich history database from standard tables to blockchain tables, and you cannot change retention times, unless you also change the credentials or connection string.

1. Go to the console and select the **Channels** tab.
2. Locate the channel that you want to modify access for. Click its **More Options** button and select **Configure Rich History**.

   The Configure Rich History dialog is displayed.
3. To enable collection of rich history data for the channel, select the **Enable Rich History** checkbox. To disable collection of rich history data for the channel, clear the **Enable Rich History** checkbox.
4. To configure the channel to collect rich history data using a different database or different settings, select **Use channel level configuration**, and then specify the settings to use.

   For more information about the rich history settings, see Enable and Configure the Rich History Database.
5. Click **Save**.

Monitor the Rich History Status

After configuring the rich history database, you can use the console to monitor the rich history replication status.

1. Go to the console and select the **Channels** tab.
2. In the channels table, click the **More Actions** button for the channel that you want to monitor, and then click **Rich History Status**.

   The Rich History Status dialog box is displayed, which includes details about replication and configuration status.
3. Click **Refresh** to display the latest status.

Limit Access to Rich History

You can use channel policies and access control lists (ACLs) to limit the organizations that can configure the rich history database and retrieve rich history status or configuration information.

By default, all organizations that have administrative access to a channel can configure rich history collection and can retrieve rich history status and configuration details. To limit this access to, for example, the founder organization, you create a channel policy and apply the policy to the resources that control access.

1. Go to the console and select the **Channels** tab.

   The **Channels** tab is displayed. The channel table contains a list of all of the channels on your network.

2. In the channel table, click the name of the channel where you want to limit access.
3. Click **Channel Policies**, and then create a signature policy that includes the organization members that will access the rich history functions.

   For more information about channel policies, see Work With Channel Policies and ACLs.

   For example, create a policy that includes only the identity of the founder organization, not the identity of any participant organizations.

4. Click **ACLs**.
5. In the Resources table, locate the resource that you want to update to use the new policy. Click **Expand** for the resource and then select the policy to assign to the resource

   The following table shows the resources that control access to rich history.

Resource	Access control
`obpadmin/ ConfigureRichHistoryChannel`	Controls configuring, enabling, and disabling rich history for a channel.
`obpadmin/ GetRichHistoryChannelStatus`	Controls retrieving rich history replication status for a channel.
`obpadmin/ GetRichHistoryChannelConfig`	Controls retrieving the current rich history configuration for a channel.

6. Click **Update ACLs**.

The rich history access is now controlled by the new policy. Organization members that are not included in the new policy will receive an error message when they attempt to access a resource that is controlled by the policy.

Rich History Database Tables and Columns

The rich history database contains three tables for each channel: history, state, and latest height. You'll query the history and state tables when you create analytics about your chaincodes' ledger transactions. If you've chosen to select any of the transaction details when enabling the rich history, an additional table will be created with the transaction details.

History Table

The *<instanceName><channelName>*_hist table contains ledger history. The data in this table tells you the chaincode ID, key used, if the transaction was valid, the value assigned to the key, and so on.

Note that the **value** and **valueJson** columns are used in a mutually exclusive way. That is when a key value is valid json, then the value is set into the **valueJson** column. Otherwise the value is set in the **value** column. The **valueJson** column is set up as a json column in the database, which means users can query that column using the usual Oracle JSON specific extensions.

If configured, private data is also stored in this table. For private data, the chaincode ID uses the following format: <*chaincodeName*>$$<*collectionName*>.

Column	Datatype
chaincodeId	VARCHAR2 (256)
key	VARCHAR2 (1024)
txnIsValid	NUMBER (1)
value	VARCHAR2 (4000)
valueJson	CLOB
blockNo	NUMBER NOT NULL
txnNo NUMBER	NOT NULL
txnId	VARCHAR2 (128)
txnTimestamp	TIMESTAMP
txnIsDelete	NUMBER (1)

State Table

The <*instanceName*><*channelName*>_state table contains data values replicated from the state database. You'll query the state table when you create analytics about the state of the ledger.

Note that the **value** and **valueJson** columns are used in a mutually exclusive way. That is when a key value is valid json, then the value is set into the **valueJson** column. Otherwise the value is set in the **value** column. The **valueJson** column is set up as a json column in the database, which means users can query that column using the usual Oracle JSON specific extensions.

Column	Datatype
chaincodeId	VARCHAR2 (256)
key	VARCHAR2 (1024)
value	VARCHAR2 (4000)
valueJson	CLOB
blockNo	NUMBER
txnNo	NUMBER

Latest Height Table

The <*instanceName*><*channelName*>_last table is used internally by Oracle Blockchain Platform to track 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. You can't query this database for analytics.

Transaction Details Table

The <*instanceName*><*channelName*>_more table contains attributes related to committed transactions. When enabling the rich history database, you can select which of these attributes you want to record in this table. The transaction details table only captures information about

endorser transactions - not configuration transactions or any other kind of Hyperledger Fabric transactions.

Column	Datatype
CHAINCODEID	VARCHAR2 (256)
BLOCKNO	NUMBER
TXNNO	NUMBER
TXNID	VARCHAR2(128)
TXNTIMESTAMP	TIMESTAMP
SUBMITTERCN	VARCHAR2(512)
SUBMITTERORG	VARCHAR2(512)
SUBMITTEROU	VARCHAR2(512)
CHAINCODETYPE	VARCHAR2(32)
VALIDATIONCODENAME	VARCHAR2(32)
ENDORSEMENTS	CLOB
INPUTS	CLOB
EVENTS	CLOB
RESPONSESTATUS	NUMBER(0)
RESPONSEPAYLOAD	VARCHAR2(1024)
RWSET	CLOB
BLOCKCREATORCN	VARCHAR2(512)
BLOCKCREATORORG	VARCHAR2(512)
BLOCKCREATOROU	VARCHAR2(512)
CONFIGBLOCKNUMBER	NUMBER(0)
CONFIGBLOCKCREATORCN	VARCHAR2(512)
CONFIGBLOCKCREATORORG	VARCHAR2(512)
CONFIGBLOCKCREATOROU	VARCHAR2(512)

Note

- Organization (ORG) and organization unit (OU) are driven by identity certificates, which implies that they may be assigned to multiple values. They are captured as a comma separated list in the table's values.
- For identities, the table includes information only about the "Subject" portion of the certificates, not the "Issuer" one.
- The `RWSET` column contains operations on all chaincodes (in the same ledger) performed during endorsement. As such, you will typically see both lscc read operations and the actual chaincode namespace operations.

A

Best Practices for Resilience

This topic outlines best practices to maintain continued operation in the event that a single virtual machine (VM) of an Enterprise shape instance of Oracle Blockchain Platform is taken offline for maintenance or fails unexpectedly.

The following steps apply only to instances of Oracle Blockchain Platform based on the Enterprise shape (not the Standard shape). Additionally, this guidance applies only to the following deployment models:

- A single founder organization
- A founder with one or more participant organizations

Because instances of Oracle Blockchain Platform based on the Standard shape run all components (peers, orderers, and platform services) on a single VM, the entire instance becomes unavailable if a VM fails or must be stopped for maintenance. There is no isolation across availability domains or fault domains. Do not use instances based on the Standard shape for production environments.

Instead, use instances based on the Enterprise shape for production environments. Those instances provide independent scaling of Hyperledger Fabric components and higher capacity configurations. Enterprise shape instances automatically distribute peers, orderers, and platform components across availability domains and fault domains to provide infrastructure-level fault isolation. To take advantage of this behavior and to ensure high availability, use the best practices described in the following sections.

Endorsement Policies

For a network with a single (founder) organization, use endorsement policies that allow any available peer to endorse transactions, such as `OR('Founder.member')` or `OutOf(1, 'Founder.member')`. Deploy at least two peers for the founder organization.

For a network with a founder and one participant organization, typically you can use the following policy: `OutOf(1, 'Founder.member', 'Participant1.member')`. For a stricter configuration, only if redundancy exists, you can use `OutOf(2, 'Founder.member', 'Participant1.member')`. Avoid strict policies such as `AND('Founder.member', 'Participant1.member')` unless both organizations have sufficient peer redundancy.

For more information, see Specify an Endorsement Policy.

Peer Deployment

Deploy at least two peers per organization. Oracle Blockchain Platform automatically distributes peers across availability domains and fault domains to ensure that at least one peer remains available after a VM failure.

Private Data Collections

If you use private data collections, set the **Peers Required** value (`requiredPeerCount`) to one or greater, and set the **Max Peer Count** value (`maxPeerCount`) to two or greater in the definition of the private data collection. Ensure that at least two peers are members of each private data collection and that private data is committed across at least two peers. The **Peers Required**

value is the minimum number of peers (excluding the endorsing peer) that must successfully receive the private data before the transaction proposal is considered complete.

For cross-organization private data collections, configure the required peer count to be greater than the number of peers from a single organization and ensure distribution across multiple organizations and VMs.

For more information, see Add Private Data Collections.

Raft Ordering Service

Use the default three-node Raft ordering service. Oracle Blockchain Platform distributes orderers across availability domains and fault domains, which allows the ordering service to handle a single node failure.

Anchor Peers

In networks with multiple organizations, configure at least one anchor peer per organization. For improved resilience, configure at least two anchor peers per organization. Anchor peers are needed only when more than one Oracle Blockchain Platform organization instance exists in the network, as they enable gossip-based communication and peer discovery across different organizations.

For more information, see Add an Anchor Peer.

Chaincode Deployment

To ensure continuity if a peer becomes unavailable, install and approve chaincode on at least two peers per organization. You can spread these deployments across Oracle Blockchain Platform instances in a network if your privacy requirements allow for that.

Client Connectivity

Configure client applications to never latch to specific peers. Instead, allow the underlying client library to deal with peer selection.

Fallback State Database

The state database is stored on each peer for all channels that the peer is joined to. If a VM fails, the local state database on that peer becomes unavailable until the peer is restored. Oracle Blockchain Platform supports a hybrid state database model, where an external Oracle Database acts as a fallback (secondary) state database. When the fallback state database is enabled, state data is persisted outside the peer VM in a database that can take over as the primary state database as needed, which improves durability and recovery.

Complete the following steps to use this function to improve resilience.

- Enable the fallback state database for production workloads or critical workloads
- Deploy Oracle Database independently of the peer VMs.
- Configure Oracle Database for high availability.

By taking these steps you can ensure that state data is not tied to the life cycle of a single VM. The fallback state database works in complement with peer redundancy for scenarios where a single VM fails.

For more information, see Create the Fallback State Database.

B

Node Configuration

This topic contains information to help you understand and configure your nodes. Each node type has different configuration options.

Topics:

CA Node Attributes

A certificate authority (CA) node keeps track of identities and certificates on the blockchain network.

Only Administrators can change a node's attributes. If you've got User privileges, then you can view a node's attributes.

Table B-1 CA Node Attributes

Attribute	Description	Default Value
Fabric CA ID	This is the identifier or name that Oracle Blockchain Platform assigned the node when it created it. You can't modify this ID.	ca
Listen Port	This is the listening port that Oracle Blockchain Platform assigned to the node. You can't change the port number.	Specific to your organization.
Max Enrollments	Use this field to determine how many times the CA server allows a password to be used for enrollment on the network. Consider the following options: • -1 — The server allows a password to be used an unlimited number of times for enrollment.	-1
Log Level	Specify the log level that you want to use for the node. Oracle suggests that for development or testing, you use DEBUG. And that for production, you use INFO.	INFO

Console Node Attributes

The console node manages the performance of the console.

Only Administrators can change a node's attributes. If you've got User privileges, then you can view a node's attributes.

Table B-2 Console Node Attributes

Attribute	Description	Default Value
Console ID	This is the identifier or name that Oracle Blockchain Platform assigned the node when it created it.	console
Local MSP ID	This is the assigned MSP ID for your organization. You can't modify this ID.	NA
Listen Port	This is the listening port that Oracle Blockchain Platform assigned to the node. You can't change the port number.	Specific to your organization.
Log Level	Specify the log level that you want to use for the node. Oracle suggests that for development or testing, you use DEBUG. And that for production, you use ERROR.	INFO
Request Timeout (s)	Specify the maximum amount of time in seconds that you want the console to attempt to contact the nodes before timing out.	600

Orderer Node Attributes

An orderer node collects transactions from peer nodes, bundles them, and submits them to the blockchain ledger. The node's attributes determine how the node performs and behaves on the network.

Only Administrators can change a node's attributes. If you've got User privileges, then you can view a node's attributes.

Table B-3 Orderer Node — General Attributes

Attribute	Description	Default Value
Orderer ID	This is the identifier or name that Oracle Blockchain Platform assigned the node when it created it.	orderer<number-partition>
Local MSP ID	This is the assigned MSP ID for your organization. You can't modify this ID.	NA

Table B-3 (Cont.) Orderer Node — General Attributes

Attribute	Description	Default Value
Listen Port	This is the listening port that Oracle Blockchain Platform assigned to the node. You can't change the port number.	Specific to your organization.
Log Level	Specify the log level that you want to use for the node. Oracle suggests that for development or testing, you use DEBUG. And that for production, you use ERROR.	INFO

Table B-4 Orderer Node — Advanced Attributes — Raft/Cluster tab

Attribute	Description	Default Value
SendBufferSize	The maximum number of messages in the egress buffer. Consensus messages are dropped if the buffer is full, and the transaction messages are waiting for space to be freed.	10
DialTimeout in seconds	The maximum duration of time after which connection attempts are considered as failed.	5
RPCTimeout in seconds	The maximum duration of time after which RPC attempts are considered as failed.	7
Replication/BufferSize in bytes	The maximum number of bytes that can be allocated for each in-memory buffer used for block replication from other cluster nodes.	20971520
Replication/ BackgroundRefreshInterval in minutes	The time between two consecutive attempts to replicate existing channels that this node was added to, or channels that this node failed to replicate in the past.	5
Replication/RetryTimeout in seconds	The maximum duration the ordering node will wait between two consecutive attempts.	5
Replication/PullTimeout in seconds	The maximum duration the ordering node will wait for a block to be received before it aborts.	5
Consensus/EvictionSuspicion in minutes	The threshold that a node will start suspecting its own eviction if it has been leaderless for this period of time.	2

Peer Node Attributes

A peer node reads, endorses, and writes transactions to the blockchain ledger. The node's attributes determine how the node performs and behaves on the network.

Only Administrators can change a node's attributes. If you've got User privileges, then you can view a node's attributes.

Table B-5 Peer Node — General Attributes

Attribute	Description	Default Value
Peer ID	This is the identifier or name that Oracle Blockchain Platform assigned the node when it created it.	peer0
Local MSP ID	This is the assigned MSP ID for your organization. You can't modify this ID.	Specific to your organization.
Role	Specifies if the peer's role is Member or Admin. In most cases this field displays Member. This role is used by the chaincode's endorsement policy. The endorsement policy specifies the MSP that must validate the identity of the signer peer and the signer peer's role. The Admin role is normally assigned in situations where you want to further protect sensitive operations and make sure that those operations are endorsed by specific peers. The peers created with your instance were assigned the Member role.	Member
Listen Port	This is the listening port that Oracle Blockchain Platform assigned to the node. You can't change the port number.	Specific to your organization.
Log Level	Specify the log level that you want to use for the node. Oracle suggests that for development or testing, you use DEBUG. And that for production, you use ERROR.	INFO
Alias	Optionally, enter text to further identify the peer beyond the peer ID.	NA

Table B-6 Peer Node — Advanced Attributes — Gossip tab

Attribute	Description	Default Value
Bootstrap Peers	Provide the service name address and port that the peer uses to contact other peers during startup. This endpoint must match the endpoints of the peers in the same organization.	NA
Max Block Count to Store	Enter the maximum number of blocks to store in memory.	10
Max Propagation Burst Latency in milliseconds	Enter how many milliseconds between message pushes.	10
Max Propagation Burst Size	Enter the number of messages to be stored until a push remote peer is triggered.	10
Propagate Iterations	Enter the number of times a message is pushed to the peers.	1
Max Connection Attempts	Enter the maximum number of attempts to make when connecting to a peer.	120
Message Expiration Factor	Enter the message expiration factor for alive messages.	20
Propagate Peer Number	Enter how many peers to send messages to.	3
Pull Interval in seconds	Enter how many seconds between pull phases.	4
Pull Peer Number	Enter the number of peers to pull from.	3
Request State Info Interval in seconds	Enter how often to pull state information messages from the peers.	4
Publish State Info Interval in seconds	Enter how often to send state information messages to the peers.	4
Publish Cert Period in seconds	Enter how many seconds from startup that certificates are included in alive messages.	10
Dial Timeout in seconds	Enter how many seconds before dial times out.	3
Connect Timeout in seconds	Enter how many seconds until the connection times out.	2
Receive Buffer Size	Enter the size of the buffer for received messages.	20
Send Buffer Size	Enter the size of the buffer for sending messages.	200
Digest Wait Time in seconds	Enter how many seconds to wait before the pull engine processes incoming digests.	1
Request Wait Time in seconds	Enter how many seconds to wait before the pull engine removes incoming nonce.	1,500

Table B-6 (Cont.) Peer Node — Advanced Attributes — Gossip tab

Attribute	Description	Default Value
Response Wait Time in seconds	Enter how many seconds that the pull engine waits before it terminates the pull.	2
Alive Time Interval in seconds	Enter how often to check alive time.	5
Alive Expiration Timeout in seconds	Enter how many seconds to wait before the alive expiration times out.	25
Reconnect Interval in seconds	Enter how many seconds to wait before reconnecting.	25
Skip Block Verification	Click to skip block verification.	Not selected

Table B-7 Peer Node — Advanced Attributes — Gossip/Election tab

Attribute	Description	Default Value
Membership Sample Interval in seconds	How often in seconds the peer checks its stability on the network.	1
Leader Alive Threshold in seconds	The number of seconds to elapse before the last declaration message is sent and before the peer determines leader election.	10
Leader Election Duration in seconds	The number of seconds to elapse after the peer sends the propose message and declares itself leader.	5
Leader	A channel's leader peer receives blocks and distributes them to the other peers within the cluster. Specify the mode that you want the peer to use to determine a leader. • **OrgLeader** — Select this option to use static leader mode and make the peer the organization leader. If you select this option and then add more peers to the channel, then you must set all peers to **OrgLeader**. • **UseLeaderElection** — Select this option to use dynamic leader election on the channel. Before an active leader is selected for the organization, the system must run the configuration transaction to add the organization to the channel, and then the system updates the new peers with the configuration transaction.	OrgLeader

Table B-8 Peer Node — Advanced Attributes — Event Service tab

Attribute	Description	Default Value
Buffer Size	Enter the maximum number of events that the buffer can contain. The system won't send the events that exceed this number.	100
Timeout in milliseconds	Enter in milliseconds the maximum time allowed for the business network to send an event.	10

Table B-9 Peer Node — Advanced Attributes — Chaincode tab

Attribute	Description	Default Value
Startup timeout in seconds	Enter in seconds the maximum time to wait between when the container starts and the registry responds.	300
Install timeout in seconds	Enter in the seconds the maximum time to wait for a chaincode to complete the installation process before timing out.	300
Execute timeout in seconds	Enter in seconds the maximum time that a chaincode attempts to execute before timing out.	30
Mode	Displays how the system runs the chaincode. This value is always net.	net
Keepalive in seconds	If you're using a proxy for communication, then enter in seconds the maximum amount of time to keep the connection between a peer and the chaincode alive.	0
Log Level	Specify the log level that you want to use for all loggers in the chaincode container. Oracle suggests that for development or testing, you use DEBUG. And that for production, you use ERROR.	INFO
Shim Level	Specify the log level that you want to use for the shim logger.	WARNING

REST Proxy Node Attributes

A REST proxy node lets you query or call a chaincode via the RESTful protocol. The node's attributes determine how the node performs on the network and which channel, chaincode, and peers are used in the transactions performed by the node.

Only Administrators can change a node's attributes. If you've got User privileges, then you can view a node's attributes.

Table B-10 REST Proxy Node Attributes

Attribute	Description	Default Value
REST Proxy Name	This is the identifier or name that Oracle Blockchain Platform assigned the node when it created it. You can't modify this ID.	restproxy
Proposal Wait Time (ms)	Enter the number of milliseconds that the node waits for completion of the proposal process. If this number is exceeded, then the transaction times out.	60,000
Transaction Wait Time (ms)	Enter the number of milliseconds that the node waits for execution after the transaction is submitted. If this number is exceeded, then the transaction times out.	300,000
Log Level	Specify the log level that you want to use for the node. Oracle suggests that for development or testing, you use DEBUG. And that for production, you use WARNING or ERROR.	INFO
Transaction Event Logging	If you specify a log level of INFO, you can also enable or disable transactional event logging. Transaction events are not logged when the log level is WARNING or ERROR, and they are always logged when the log level is DEBUG.	Disabled

C

Using the Fine-Grained Access Control Library Included in the Marbles Sample

Hyperledger Fabric provides fine-grained access control to many of the management functions. Oracle Blockchain Platform provides a marbles sample package on the Developer Tools tab of the console, implementing a library of functions that chaincode developers can use to create access control lists for chaincode functions. It currently supports only the Go language.

Background

The goal of this sample access control library is to provide the following functions:

- Provides a mechanism to allow you to control which users can access particular chaincode functions. The list of users and their entitlements is dynamic and shared across chaincodes.
- Provides access control checks so that a chaincode can check the access control list easily.
- At chaincode deployment time, lets you to populate the list of resources and access control lists with your initial members. An access control list must be provided to authorize users to perform access control list operations.

Download the Sample

On the **Developer Tools** tab, open the **Samples** pane. Click the download link under **Marbles with Fine-Grained ACLs**. This package contains three sub-packages:

- `Fine-GrainedAccessControlLibrary.zip`:
  The fine-grained access control library. It contains functions in Go which can be used by chaincode developers to create access control lists for chaincode functions.
- `fgACL_MarbleSampleCC.zip`:
  The marbles sample with access control lists implemented. It includes a variety of functions to let you examine how to work with fine-grained access control lists, groups and resources to restrict functions to certain users/identities.
- `fgACL-NodeJSCode.zip`:
  Node.js scripts which use the Node.js SDK to run the sample. `registerEnrollUser.js` can be used to register new users with the Blockchain Platform. `invokeQueryCC.js` can be used to run transactions against a Blockchain Platform instance.

Terminology and Acronyms

Term	Description
Identity	An X509 certificate representing the identity of either the caller or the specific identity the chaincode wants to check.

Term	Description
Identity Pattern	A pattern that matches one or more identities. The following patterns are suggested: • X.509 Subject Common Name – CN • X.509 Subject Organizational Unit – OU • X.509 Subject Organization – O • Group as defined in this library – GRP • Attribute – ATTR The format for a pattern is essentially just a string with a prefix. For example, to define a pattern that matches any identity in organization "example.com", the pattern would be "%O%example.com".
Resource	The name of anything the chaincode wants to control access to. To this library it is just a named arbitrary string contained in a flat namespace. The semantics of the name are completely up to the chaincode.
Group	A group of identity patterns.
ACL	Access Control List: a named entity that has a list of identity patterns, a list of types of access such as "READ", "CREATE", "INVOKE", "FORWARD", or anything the chaincode wants to use. This library will use access types of CREATE, READ, UPDATE, and DELETE (standard CRUD operations) to maintain its information. Other than those four as they relate to the items in this library, they are just strings with no implied semantics. An application may decide to use accesses of "A", "B", and "CUSTOM".

Fine-Grained Access Control Library Functions

The library package provides the following functions for resources, groups and ACLs as well as global functions.

Global Functions

Function	Description
`Initialization(identity *x509.Certificate, stub shim.ChaincodeStubInterface) (error) (error)`	When the chaincode is deployed, the `Initialization` function is called. This function initializes the world state with some built-in access control lists. These built-in lists are used to bootstrap the environment. There must be access control on who can create resources, groups, and ACLs. If the identity is nil, then the function uses the caller's identity. After the bootstrap process is done, the following entities are created: • A resource named `.Resources`. A corresponding ACL named `.Resources.ACL` will be created with a single identity pattern in it of the form "%CN%bob.smith@oracle.com", using the actual common name, and the access will be CREATE, READ, UPDATE, and DELETE access. • A group named `.Groups`. A corresponding ACL named `.Groups.ACL` will be created with a single identity pattern in it of the form "%CN%bob.smith@oracle.com", using the actual common name, and the access will be CREATE, READ, UPDATE, and DELETE access. • An ACL named `.ACLs`. A corresponding ACL control list named `.ACLs.ACL` will be created with a single identity pattern in it of the form "%CN%bob.smith@oracle.com", using the actual common name, and the access will be CREATE, READ, UPDATE, and DELETE access.
`NewGroupManager(identity *x509.Certificate, stub shim.ChaincodeStubInterface) (*GroupManager, error)`	Gets the group manager that's used for all group related operations. Identity: the default identity for related operation. If it's nil, then the function uses the caller's identity.
`NewACLManager(identity *x509.Certificate, stub shim.ChaincodeStubInterface) (*ACLManager, error)`	Gets the ACL manager that's used for all ACL related operations. Identity: the default identity for related operation. If it's nil, then the function uses the caller's identity.
`NewResourceManager(identity *x509.Certificate, stub shim.ChaincodeStubInterface) (*ResourceManager, error)`	Gets the resource manager that's used for all resource related operations. Identity: the default identity for related operation. If it's nil, then the function uses the caller's identity.

Access Control List (ACL) Functions

Definition of `ACL` structure:

```
type ACL struct {
  Name string
  Description string
  Accesses []string  // CREATE, READ, UPDATE, and DELETE, or whatever the end-
user defined
  Patterns []string    // identities
  Allowed bool          // true means allows access.
  BindACLs []string  // The list of ACL , control who can call the APIs of
this struct
}
```

- **Accesses:** The Accesses string is a list of comma-separated arbitrary access names and is completely up to the application except for four: CREATE, READ, UPDATE, and DELETE. These access values are used in maintaining the fine-grained access control. Applications can use their own access strings such as `"register"`, `"invoke"`, or `"query"`, or even such things as access to field names such as `"owner"`, `"quantity"`, and so on.
- **Allowed:** Allowed determines whether identities that match a pattern are allowed access (true) or prohibited access (false). You can have an access control list that indicates Bob has access to `"CREATE"`, and another one that indicates group Oracle (of which Bob is a member) is prohibited from `"CREATE"`. Whether Bob has access or not depends upon the order of the access control lists that are associated with the entity in question.
- **BindACLs:** The BindACLs parameter forms the initial access control list.

ACL functions:

Function	Description
`Create(acl ACL, identity *x509.Certificate) (error)`	Creates an ACL. Duplicate named ACLs are not allowed. To create an ACL, the identity must have CREATE access to the bootstrap resource named ".ACLs". If identity is nil, the default identity specified in `newACLManager()` is used.
`Get(aclName string, identity *x509.Certificate) (ACL, error)`	Gets a named ACL. The identity must have READ access to the named ACL. If identity is nil, the default identity specified in `newACLManager()` is used.
`Delete(aclName string, identity *x509.Certificate) (error)`	Deletes a specified ACL. The identity must have DELETE access to the named ACL. If identity is nil, the default identity specified in `newACLManager()` is used.
`Update(acl ACL, identity *x509.Certificate) (error)`	Updates an ACL. The identity must have UPDATE access to the named resource, and the named ACL must exist. If identity is nil, the default identity specified in `NewACLManager()` is used.

Function	Description
`AddPattern(aclName string, pattern string, identity *x509.Certificate) (error)`	Adds a new identity pattern to the named ACL. The identity must have UPDATE access to the named ACL. If identity is nil, the default identity specified in `newACLManager()` is used.
`RemovePattern(aclName string, pattern string, identity *X509Certificate) (error)`	Removes the identity pattern from the ACL. The identity must have UPDATE access to the named ACL. If identity is nil, the default identity specified in `newACLManager()` is used.
`AddAccess(aclname string, access string, identity *X509Certificate) (error)`	Adds a new access to the named ACL. The identity must have UPDATE access to the named ACL. If identity is nil, the default identity specified in `newACLManager()` is used.
`RemoveAccess(aclName string, access string, identity *X509Certificate) (error)`	Removes the access from the ACL. The identity must have UPDATE access to the named ACL. If identity is nil, the default identity specified in `newACLManager()` is used.
`UpdateDescription(aclName string, newDescription string, identity *X509Certificate) (error)`	Updates the description. The identity must have UPDATE access to the named ACL. If identity is nil, the default identity specified in `newACLManager()` is used.
`AddBeforeACL(aclName string, beforeName string, newBindACL string, identity *X509Certificate) (error)`	Adds a bind ACL before the existing named ACL. If the named ACL is empty or not found, the ACL is added to the beginning of the bind ACL list. The identity must have UPDATE access to the named ACL. If the identity is nil, the default identity specified in `newACLManager()` is used.
`AddAfterACL(aclName string, afterName string, newBindACL string, identity *X509Certificate) (error)`	Adds a bind ACL after the existing named ACL. If the named ACL is empty or not found, the ACL is added to the end of the bind ACL list. The identity must have UPDATE access to the named ACL. If the identity is nil, the default identity specified in `newACLManager()` is used.
`RemoveBindACL(aclName string, removeName string, identity *X509Certificate) (error)`	Removes the `removeName` ACL from the bind ACL list. The identity must have UPDATE access to the named ACL. If the identity is nil, the default identity specified in `newACLManager()` is used.
`GetAll(identity *x509.Certificate) ([]ACL, error)`	Get all the ACLs. The identity must have READ access to the named ACL. If the identity is nil, the default identity specified in `newACLManager()` is used.

Group Functions

Definition of `Group` structure:

```
type Group struct {
    Name string
    Description string
```

```
    Members []string     // identity patterns, except GRP.
    BindACLs []string    // The list of ACLs, controls who can access this
group.
}
```

Definition of `GroupManager` functions:

Function	Description
`Create(group Group, identity *x509.Certificate) (error)`	Creates a group. The identity must have CREATE access to bootstrap group `.Group`. If identity is nil, the default identity specified in `NewGroupManager()` is used.
`Get(groupName string, identity *x509.Certificate) (Group, error)`	Gets a specified group. The identity must have READ access to this group. If identity is nil, the default identity specified in `NewGroupManager()` is used.
`Delete(groupName string, identity *x509.Certificate) (error)`	Deletes a specified group. The identity must have DELETE access to this group. If identity is nil, the default identity specified in `NewGroupManager()` is used.
`AddMembers(groupName string, member []string, identity *x509.Certificate) (error)`	Adds one or more members to the group. The identity must have UPDATE access to this group. If identity is nil, the default identity specified in `NewGroupManager()` is used.
`RemoveMembers(groupName string, member []string, identity *x509.Certificate) (error)`	Removes one or more member from a group. The identity must have UPDATE access to this group. If identity is nil, the default identity specified in `NewGroupManager()` is used.
`UpdateDescription(groupName string, newDes string, identity *x509.Certificate) (error)`	Updates the description. The identity must have UPDATE access to this group. If identity is nil, the default identity specified in `NewGroupManager()` is used.
`AddBeforeACL(groupName string, beforeName string, aclName string, identity *x509.Certificate) (error)`	Adds a bind ACL to the group before the existing named ACL. If the named ACL is empty or not found, the ACL is added to the beginning of the list of bind ACL for the resource. The identity must have UPDATE access to the named group. If identity is nil, the default identity specified in `NewGroupManager()` is used.
`AddAfterACL(groupName string, afterName string, aclName string, identity *x509.Certificate) (error)`	Adds a bind ACL to the group after the existing named ACL. If the named ACL is empty or not found, the ACL is added to the end of the list of bind ACLs for the group. The identity must have UPDATE access to the named group. If the identity is nil, the default identity specified in `NewGroupManager()` is used.
`RemoveBindACL(groupName string, aclName string, identity *x509.Certificate) (error)`	Removes the named ACL from the bind ACL list of the named group. The identity must have UPDATE access to the named group. If the identity is nil, the default identity specified in `NewGroupManager()` is used.

Function	Description
`GetAll(identity *x509.Certificate) ([]Group, error)`	Gets all groups. The identity must have READ access to these groups. If identity is nil, the default identity specified in `NewGroupManager()` is used.

Resource Functions

Definition of `Resource` structure:

```
type Resource struct {
     Name string
     Description string
     BindACLs []string      // The name list of ACL, controls who can access
this resource
}
```

Resource Functions:

Fuction	Description
`Create(resource Resource, identity *x509.Certificate) (error)`	Creates a resource. Duplicate named resources are not allowed. The identity must have CREATE access to the `.Resources` bootstrap resource. If identity is null, the default identity specified in `NewResourceManager()` is used.
`Get(resName string, identity *x509.Certificate) (Resource, error)`	Gets a specified resource. The identity must have READ access to the resource. If identity is null, the default identity specified in `NewResourceManager()` is used.
`Delete(resName string, identity *x509.Certificate) (error)`	Deletes a named resource. The identity must have DELETE access to the named resource. If identity is null, the default identity specified in `NewResourceManager()` is used.
`UpdateDescription(resourceName string, newDes string, identity *x509.Certificate) (error)`	Updates the description. The identity must have UPDATE access to this resource. If identity is nil, the default identity specified in `NewResourceManager()` is used.
`AddBeforeACL(resourceName string, beforeName string, aclName string, identity *x509.Certificate) (error)`	Adds a bind ACL to the resource before the existing named ACL. If the named ACL is empty or not found, the ACL is added to the beginning of the list of bind ACL for the resource. The identity must have UPDATE access to the named resource. If identity is nil, the default identity specified in `NewResourceManager()` is used.

Fuction	Description
`AddAfterACL(resourceName string, afterName string, aclName string, identity *x509.Certificate) (error)`	Adds a bind ACL to the resource after the existing named ACL. If the named ACL is empty or not found, the ACL is added to the end of the list of bind ACL for the resource. The identity must have UPDATE access to the named resource. If the identity is nil, the default identity specified in `NewResourceManager()` is used.
`RemoveBindACL(resourceName string, aclName string, identity *x509.Certificate) (error)`	Removes the named ACL from the bind ACL list of the named resource. The identity must have UPDATE access to the named resource. If the identity is nil, the default identity specified in `NewResourceManager()` is used.
`CheckAccess(resName string, access string, identity *x509.Certificate) (bool, error)`	Checks whether the current user has the specified access to the named resource. If the identity is nil, the default identity specified in `NewResourceManager()` is used.
`GetAll(identity *x509.Certificate) ([]Resource, error)`	Gets all resources. The identity must have READ access to these resources. If identity is nil, the default identity specified in `NewResourceManager()` is used.

Example Walkthough Using the Fine-Grained Access Control Library

This topic provides some examples of how the library and chaincode can be used. These examples all assume that the `Init()` function has been called to create the bootstrap entities and the caller of `Init()` and `invoke()` is `"%CN%frank.thomas@example.com"`. The normal flow in an application is to create some initial access control lists that will be used to grant or deny access to the other entities.

Initialization

Call the `Initialization()` function to create bootstrap entities when deploying chaincodes. For example:

```
import "chaincodeACL"
func (t \*SimpleChaincode) Init(nil, stub shim.ChaincodeStubInterface)
pb.Response
{
        err := chaincodeACL.Initialization(stub)
}
```

Create an ACL

```
import "chaincodeACL"
...
{
```

```
**ACLMgr**  := chaincodeACL.NewACLManager(nil, stub) // Not specify identity,
use caller's identity as default.

// Define a new ACL
**newACL**  := chaincodeACL.ACL{

    "AllowAdmins",   // ACL name
    "Allow administrators full access",  // Description
    []string{"CREATE","READ","UPDATE","DELETE"},    // Accesses allowed or not
    true, // Allowed
    []string{"%CN%bob.dole@example.com","%OU%example.com,"%GRP%admins"}, //
Initial identity patterns
    ".ACLs.acl", // Start with bootstrap ACL

}

// Add this ACL with default identity (caller's identify here)
err :=  **ACLMgr**.Create( **newACL** , nil)

}
```

You can use the new ACL to modify who can perform certain operations. First, add this new ACL to the bootstrap group `.Groups` to allow any administrator to create a group.

Add an ACL to a group

```
import "chaincodeACL"
…
{

  **groupMgr**  := chaincodeACL.NewGroupManager(nil, stub) // Not specify
identity, use caller's identity as default.
  err :=  **groupMgr**.AddAfterACL(

    ".Groups",     // Bootstrap group name
    ".Groups.ACL", // Which ACL to add after
    "AllowAdmins", // The new ACL to add
    nil            // with default identity that's frank.thomas

)

}
```

This adds the `AllowAdmins` ACL to the bootstrap group `.Groups` after the initial bootstrap ACL. Thus this ensures that Frank Thomas can still complete operations on the `.Groups` group because the ACL that grants Frank permission is first in the list. Now, anyone who matches the `AllowAdmins` ACL can complete CREATE, READ, UPDATE, or DELETE operations (they can now create groups).

Create a group

Administrators can now create a group.

```
import "chaincodeACL"
...
```

```
{

...
  // Define a new group.
  **newGroup**  := chaincodeACL.Group{

      "AdminGrp",   // Name of the group
      "Administrators of the app",   // Description of the group

{"%CN%jill.muller@example.com","%CN%ivan.novak@example.com","%ATTR%role=admin"
},
      []string{"AllowAdmins"},   // The ACL for the group

    }

  **groupMgr**  := chaincodeACL.NewGroupManager(nil, stub)   // Not specify
identity, use caller's identity as default.
  err :=  **groupMgr**.Create( **newGroup** , bob\_garcia\_certificate)   //
Using a specific certificate

...
}
```

This call is using an explicit identity (Bob Garcia, by using his certificate) to attempt to create a group. Because Bob Garcia matches a pattern in the `AllowAdmins` ACL and members of that ACL can perform CREATE operations on the bootstrap group `.Groups`, this call will succeed. Had Jim Silva, who was not in organization unit `example.com` nor in the group `AdminGrp` (which still doesn't exist), had his certificate passed as the last argument, the call would fail as he doesn't have the appropriate permissions. This call creates a group called "`AdminGrp`" with initial members of the group being jill.muller@example.com and ivan.novak@example.com or anyone with the attribute (ABAC) role=admin.

Create a resource

```
import "chaincodeACL"
...
{

  ...
  **newResource**  :=  **chaincodeACL**.Resource{

      "transferMarble", // Name of resource to create

      "The transferMarble chaincode function", // Description of the resource

      []string{"AllowAdmins"}, // Single ACL for now allowing administrators

  }

  **resourceMgr**  :=  **chaincodeACL**.NewResourceManager(nil, stub)  // Not
specify identity, use caller's identity as default.
  err :=  **resourceMgr**.Create(resourceMgr, nil)   // Using caller's
certificate
```

```
  ...
}
```

This creates a resource named `transferMarble` that the application might use to control access to the `transferMarble` chaincode function. The access is currently limited by the `AllowAdmins` access control list.

Check access for a resource

You can use this new resource in your chaincode to allow only administators to transfer a marble, by modifying the `invoke()` method of the Marbles chaincode as shown in the following code:

```
import "chaincodeACL"
…
func (t \*SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface)
pb.Response {

  **resourceMgr**  :=  **chaincodeACL**.NewResourceManager(nil, stub)   //
Not specify identity, use caller's identity as default.

  function, args := stub.GetFunctionAndParameters()

  fmt.Println("invoke is running " + function)        // Handle different
functions

  if function == "initMarble" {   //create a marble

      return t.initMarble(stub, args)}

  else if function == " **transferMarble**" { //change owner of a specific
marble

    **allowed** , err : =  **resourceMgr**. **CheckAccess**
("transferMarble", "UPDATE", nil)
    if  **allowed**  == true {

      return t.transferMarble(stub, args)

    else {

      return NOACCESS

    }

    } else if function == "transferMarblesBasedOnColor" { //transfer all
marbles of a certain color
    …

    }

}
```

Fine-Grained Access Control Marbles Sample

The marbles chaincode application lets you create assets (marbles) with unique attributes (name, size, color and owner) and trade these assets with fellow participants in a blockchain network.

This sample application includes a variety of functions to let you examine how to work with access control lists and groups to restrict functions to certain users.

Overview of the Sample

The test scenario included in the sample contains the following restrictions in order to manage assets:

- Bulk transfer of red marbles is only allowed by identities having the `"redMarblesTransferPermission"` Fabric attribute.
- Bulk transfer of blue marbles is only allowed by identities having the `"blueMarblesTransferPermission"` Fabric attribute.
- Deletion of marbles is only allowed to identities with `"deleteMarblePermission"` Fabric attribute.

These restrictions are enforced by implementing the following library methods in the `fgMarbles_chaincode.go` chaincode:

- Create a fine-grained ACL group named `bulkMarblesTransferGroup`. This group will define all the identities which can transfer marbles based on color (bulk transfers):

```
createGroup(stub, []string{" bulkMarblesTransferGroup",
"List of Identities allowed to Transfer Marbles in Bulk",
"%ATTR%redMarblesTransferPermission=true,
%ATTR%blueMarblesTransferPermission=true", ".ACLs"})
```

- Create a fine-grained ACL named `redMarblesAcl` which provides bulk transfer of red marbles access to `bulkMarblesTransferGroup`:

```
createACL(stub, []string{"redMarblesAcl",
"ACL to control who can transfer red marbles in bulk",
"redMarblesTransferPermission", "%GRP%bulkMarblesTransferGroup", "true",
".ACLs"})
```

- Create a fine-grained ACL named `blueMarblesAcl` which provides bulk transfer of blue marbles access to `bulkMarblesTransferGroup`:

```
createACL(stub, []string{"blueMarblesAcl",
"ACL to control who can transfer blue marbles in bulk",
```

```
"blueMarblesTransferPermission", "%GRP%bulkMarblesTransferGroup", "true",
".ACLs"})
```

- Create a fine-grained ACL named `deleteMarbleAcl` to restrict marble deletion based on `"canDeleteMarble=true"` Fabric attribute:

```
createACL(stub, []string{"deleteMarbleAcl",
"ACL to control who can Delete a Marble",
"deleteMarblePermission", "%ATTR%deleteMarblePermission=true", "true",
".ACLs"})
```

- Create a fine-grained ACL resource named `marble`, operations on which are controlled using the various ACLs we created:

```
createResource(stub, []string{"marble",
"System marble resource",
"deleteMarbleAcl,blueMarblesAcl,redMarblesAcl,.ACLs"})
```

Pre-requisites and Setup

To run the fine-grained access control version of the marbles sample, complete these steps:

1. Download the fine-grained access control version of the marbles sample. On the **Developer Tools** page, open the **Samples** pane, and then click the download link under **Marbles with Fine-Grained ACLs**. Extract this package, which contains `.zip` files of the marbles sample (`fgACL_MarbleSampleCC.zip`), Node.js files to run the sample (`fgACL-NodeJSCode.zip`), and the fine-grained access control library (`Fine-GrainedAccessControlLibrary.zip`).
2. Generate the chaincode package that will be deployed to Oracle Blockchain Platform:
   - Extract the contents of the `fgACL_MarbleSampleCC.zip` file to the `fgACL_MarbleSampleCC` directory. The contents of the `fgACL_MarbleSampleCC` directory will be the `fgACL_Operations.go`, `fgGroups_Operations.go`, `fgMarbles_chaincode.go` ,`fgResource_Operations.go`, and `go.mod` files and the `oracle.com` directory.
   - From the command line, go to the `fgACL_MarbleSampleCC` directory, and run `GO111MODULE=on go mod vendor`. This command downloads the required dependencies and adds them to the `vendor` directory.
   - Compress all the contents (the four Go files, the `go.mod` file, and the `vendor` and `oracle.com` directories) of the `fgACL_MarbleSampleCC` directory in ZIP format. Your chaincode is ready to be deployed to Oracle Blockchain Platform.
3. Install and deploy the updated sample chaincode package (`fgACL_MarbleSampleCC.zip`) as described in Use Quick Deployment.
4. On the **Developer Tools** page, open the **Application Development** pane, and then follow the instructions to download the Node.js SDK.
5. On the **Developer Tools** page, open the **Application Development** pane, and then click **Download the development package**.
   a. Extract the development package into the same folder with the Node.js files downloaded with the sample.
   b. In the `network.yaml` file, look for the `certificateAuthorities` entry and its `registrar` entry. The administrator's password is masked (converted to `***`) in the

`network.yaml` when downloaded. Replace this password with the administrator's clear text password when running this sample.

6. Register a new identity with your Oracle Blockchain Platform instance:
   a. Create a user in IDCS (referred to as `<NewIdentity>` in the following steps) in the IDCS mapped to your tenancy.
   b. Give this user the `CA_User` application role for your instance.

Implement the Fine-Grained Access Control Marble Sample

Complete the following steps to enroll your new user and implement the ACL restrictions using the provided Node.js scripts.

1. **Enroll the new user:**

```
node registerEnrollUser.js <NewIdentity> <Password>
```

2. **Initialization:** Initialize the access control lists.

```
node invokeQueryCC.js <NewIdentity> <Password> <ChannelName>
<ChaincodeName> ACLInitialization
```

3. **Create the access control lists, groups, and resources:** This creates the ACL resources described in the overview.

```
node invokeQueryCC.js <NewIdentity> <Password> <ChannelName>
<ChaincodeName> createFineGrainedAclSampleResources
```

4. **Create your test marble resources:** This creates several test marble assets - blue1 and blue2 owned by tom, red1 and red2 owned by jerry, and green1 and green2 owned by spike.

```
node invokeQueryCC.js <NewIdentity> <Password> <ChannelName>
<ChaincodeName> createTestMarbles
```

Testing the Access Control

To test that our access control lists are allowing only the correct users to perform each function, run through some sample scenarios.

1. **Transfer a marble:** Transfer marble `blue1` from tom to jerry. Because there are no restrictions on who can transfer a single marble, this completes successfully.

```
node invokeQueryCC.js <NewIdentity> <Password> <ChannelName>
<ChaincodeName> transferMarble blue1 jerry
```

2. **Transfer a marble as the administrative user:** Transfer marble `blue1` from jerry to spike. Because there are no restrictions on who can transfer a single marble, this also completes successfully.

```
node invokeQueryCC.js <AdminIdentity> <Password> <ChannelName>
<ChaincodeName> transferMarble blue1 spike
```

3. **Get history:** Query the history of the marble named `blue1`. It returns that it was transferred first to jerry then to spike.

   ```
   node invokeQueryCC.js <AdminIdentity> <Password> <ChannelName>
   <ChaincodeName> getHistoryForMarble blue1
   ```

4. **Transfer all red marbles:** The `redMarblesAcl` ACL allows this transfer because the newly registered identity has the required `"redMarblesTransferPermission=true"` Fabric attribute, so the two red marbles will be transferred to tom.

   ```
   node invokeQueryCC.js <NewIdentity> <Password> <ChannelName>
   <ChaincodeName> transferMarblesBasedOnColor red tom
   ```

5. **Transfer all red marbles as the administrative user:** The administrative identity doesn't have the `"redMarblesTransferPermission=true"` Fabric attribute, so the `redMarblesAcl` ACL blocks this transfer.

   ```
   node invokeQueryCC.js <AdminIdentity> <Password> <ChannelName>
   <ChaincodeName> transferMarblesBasedOnColor red jerry
   ```

6. **Transfer all green marbles:** By default, only explicitly defined access is allowed. Because there isn't an ACL which allows for bulk transfer of green marbles, this fails.

   ```
   node invokeQueryCC.js <NewIdentity> <Password> <ChannelName>
   <ChaincodeName> transferMarblesBasedOnColor green tom
   ```

7. **Delete a marble:** The `deleteMarbleAcl` ACL allows this deletion because the newly registered identity has the required `"deleteMarblePermission=true"` Fabric attribute.

   ```
   node invokeQueryCC.js <NewIdentity> <Password> <ChannelName>
   <ChaincodeName> delete green1
   ```

8. **Delete a marble as the administrative user:** The `deleteMarbleAcl` ACL prevents this deletion because the administrative identity doesn't have the required `"deleteMarblePermission=true"` Fabric attribute.

   ```
   node invokeQueryCC.js < AdminIdentity > <Password> <ChannelName>
   <ChaincodeName> delete green2
   ```

Sample Files Reference

These tables list the methods available in the chaincode and application files included with the sample.

fgMarbles_chaincode.go

Function	Description
`initMarble`	Create a marble
`transferMarble`	Transfer a marble from one owner to another based on name
`createTestMarbles`	Calls `initMarble` to create sample marbles for testing purposes

Function	Description
`createFineGrainedAclSampleResources`	Creates the fine-grained access control list (ACL), groups, and resources required by our test scenario
`transferMarblesBasedOnColor`	Transfers multiple marbles of a certain color to another owner
`delete`	Delete a marble
`readMarble`	Returns all attributes of a marble based on name
`getHistoryForMarble`	Returns a history of values for a marble

fgACL_Operations.go

Methods	Parameters	Description
`getACL`	• `name`	Get a named ACL or read all ACLs. The user calling the method must have READ access to the named ACL.
`createACL`	• `name` • `description` • `accesses` • `patterns` • `allowed` • `BindACLs` • `Identity_Certificate`	To create an ACL, the user calling the method must have CREATE access to the bootstrap resource named `". ACLs"`. Duplicate named ACLs are not allowed
`deleteACL`	• `name`	The user calling the method must have DELETE access to the named ACL.
`updateACL`	• `name` • `description` • `accesses` • `patterns` • `allowed` • `BindACLs`	The user calling the method must have UPDATE access to the named resource, and the named ACL must exist.
`addAfterACL`	• `aclName` • `existingBindAclName` • `newBindAclName`	The user calling the method must have UPDATE access to the named resource, and the named ACL must exist.
`addBeforeACL`	• `aclName` • `existingBindAclName` • `newBindAclName`	The user calling the method must have UPDATE access to the named resource, and the named ACL must exist.
`addPatternToACL`	• `aclName` • `BindPattern`	The user calling the method must have UPDATE access to the named resource, and the named ACL must exist.
`removePatternFromACL`	• `aclName` • `BindPattern`	The user calling the method must have UPDATE access to the named resource, and the named ACL must exist.

Methods	Parameters	Description
`updateDescription`	• `aclName` • `newDesc`	The user calling the method must have UPDATE access to the named resource, and the named ACL must exist.
`removeBindACL`	• `aclName` • `bindAclNameToRemove`	The user calling the method must have UPDATE access to the named resource, and the named ACL must exist.
`addAccess`	• `aclName` • `accessName`	The user calling the method must have UPDATE access to the named resource, and the named ACL must exist.
`removeAccess`	• `aclName` • `accessName`	The user calling the method must have UPDATE access to the named resource, and the named ACL must exist.
`ACLInitialization`	• none	This function is used to initialize the fine-grained ACL support.

fgGroups_Operations.go

Methods	Parameters	Description
`getGroup`	• `name`	If name=`"GetAll"`, it returns all the groups the identity has access to. Otherwise, it returns the individual group details (if accessible) based on name. The user calling the method must have READ access to this group.
`createGroup`	• `name` • `description` • `patterns` • `bindACLs`	Returns `success` or `error`. The user calling the method must have CREATE access to bootstrap group `". Group"`
`deleteGroup`	• `name`	The user calling the method must have DELETE access to this group.
`addAfterGroup`	• `groupName` • `existingBindAclName` • `newBindAclName`	The user calling the method must have UPDATE access to this group.
`addBeforeGroup`	• `groupName` • `existingBindAclName` • `newBindAclName`	The user calling the method must have UPDATE access to this group.
`updateDescriptionForGroup`	• `groupName` • `newDesc`	The user calling the method must have UPDATE access to this group.
`removeBindAclFromGroup`	• `groupName` • `bindAclNameToRemove`	The user calling the method must have UPDATE access to this group.
`addMembersToGroup`	• `groupName` • `pattern`	The user calling the method must have UPDATE access to this group.

Methods	Parameters	Description
`removeMembersFromGroup`	• `groupName` • `pattern`	The user calling the method must have UPDATE access to this group.

fgResource_Operations.go

Methods	Parameters	Description
`createResource`	• `name` • `description` • `bindACLs`	The user calling the method must have CREATE access to the bootstrap resource named "`.Resources`". Duplicate named resources are not allowed.
`getResource`	• `name`	The user calling the method must have READ access to the resource
`deleteResource`	• `name`	The user calling the method must have DELETE access to the named resource
`addAfterACLInResource`	• `ResourceName` • `existingBindAclName` • `newBindAclName`	The user calling the method must have UPDATE access to this resource
`addBeforeACLInResource`	• `ResourceName` • `existingBindAclName` • `newBindAclName`	The user calling the method must have UPDATE access to this resource
`updateDescriptionInResource`	• `ResourceName` • `newDesc`	The user calling the method must have UPDATE access to this resource
`removeBindACLInResource`	• `ResourceName` • `bindAclNameToRemove`	The user calling the method must have UPDATE access to this resource
`checkResourceAccess`	• `ResourceName` • `access`	Checks whether the current user calling the method has the specified access to the named resource.

D

Run Solidity Smart Contracts with EVM on Oracle Blockchain Platform

You can run Solidity smart contracts with an Ethereum Virtual Machine (EVM) deployed as a chaincode on Oracle Blockchain Platform.

The EVM runs Solidity smart contracts in Ethereum networks. The EVM was created through the Hyperledger Burrow project and integrated into Hyperledger Fabric. This project enables you to use a Hyperledger Fabric permissioned blockchain platform to interact with Ethereum smart contracts written in an EVM-compatible language such as Solidity. See: Hyperledger Fabric EVM chaincode.

The following steps outline the process of running a Solidity smart contract on a provisioned instance of Oracle Blockchain Platform:

1. Upload the EVM chaincode package into Oracle Blockchain Platform.
2. Deploy the EVM chaincode on a channel.
3. Generate bytecode for a Solidity smart contract by using the Remix IDE.
4. Deploy the smart contract bytecode into the deployed EVM chaincode. Use the address returned from the deployment to send transactions.

Steps in this topic have been tested with `fabric-chaincode-evm:release-0.4` and may not work with other releases.

Set Up the EVM Chaincode Package

Before you can deploy the smart contract, you must set up the EVM chaincode package. Complete the following steps to create the chaincode package folder.

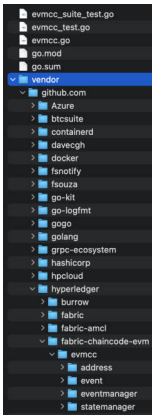
1. Download the EVM chaincode package (`.zip` file) from the following GitHub repository: Hyperledger Fabric EVM chaincode.
2. Extract the downloaded file `fabric-chaincode-evm-release-0.4.zip`.
3. In the extracted files, navigate to the `fabric-chaincode-evm-release-0.4/evmcc` directory.
4. Delete the `go.sum` file and the `vendor` directory.
5. Vendor all of the dependent Go modules by using the `go mod` command:

   ```
   go mod tidy
   go mod vendor
   ```

6. Navigate to the `fabric-chaincode-evm-release-0.4/emvcc/vendor/github.com/hyperledger` directory.
7. Create the directory `fabric-chaincode-evm/evmcc` in the `/hyperledger` directory.
8. Move the following folders from the `fabric-chaincode-evm-release-0.4/evmcc` directory to the `fabric-chaincode-evm-release-0.4/evmcc/vendor/github.com/hyperledger/fabric-chaincode-evm/evmcc` directory:

- `/address`
- `/event`
- `/eventmanager`
- `/mocks`
- `/statemanager`

The directory structure looks similar to the following screen capture when complete:



9. Compress the top-level `fabric-chaincode-evm-release-0.4/evmcc` folder in `.zip` format and rename it. The following steps use `evmcc.zip` as the example name.

Deploy EVM Chaincode on Oracle Blockchain Platform

After you create the EVM chaincode package, you deploy it on Oracle Blockchain Platform.

1. Log into the Oracle Blockchain Platform console.
2. On the **Chaincodes** tab, select **Deploy a New Chaincode**.
3. Select **Quick Deployment**, and enter the following information:
   - **Package Label:** enter a description of the chaincode package.
   - **Chaincode Language:** GoLang.
   - **Chaincode Name:** enter the name of the chaincode. For example, enter `solidityc`c.
   - **Version:** `v1`.
   - **Init-required:** leave this unselected.
   - **Channel:** select the channels where you want to install the chaincode.
   - **Chaincode source:** upload the `evmcc.zip` package that you created in the previous steps.

   For more details on the Quick Deployment wizard and restrictions on fields such as **Package Label** and **Chaincode Name**, see: Use Quick Deployment.

After you submit your information, the EVM chaincode is visible on the **Chaincodes** page and is listed as a deployed chaincode on each channel that you selected to install it on.

Create and Compile Your Solidity Smart Contract

1. Open the browser-based Remix IDE: https://remix.ethereum.org/.
2. If you already have a Solidity smart contract written, import it into Remix.

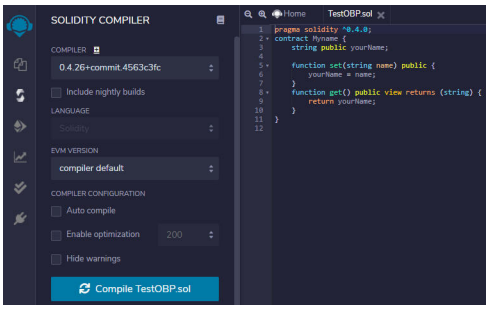
3. If you don't have a Solidity smart contract written, create a Solidity file (`.sol` file) in Remix and take one of the following steps:
   - If you're familiar with Solidity, create your own smart contract file.
   - Use the `SimpleStorage` sample code provided in the Solidity documentation: Solidity: Introduction to Smart Contracts
   - Use the sample code that is being used for this example, which takes the `string name` value as an input and prints the same value as an output string using the `set(name)` and `get()` functions.

```
pragma solidity ^0.4.0;
contract Myname {
    string public yourName;

    function set(string name) public {
        yourName = name;
    }
    function get() public view returns (string) {
        return yourName;
    }
}
```

   You might see an error message about the default compiler version not matching the version that you've specified in your smart contract.

4. Compile your smart contract. Open the Solidity Compiler panel in Remix, ensure that your smart contract tab is open to select it as the file being compiled, set the compiler version to the most recent 4.X version, and then select **Compile**.



5. After the file is compiled, select the **Bytecode** icon to copy the bytecode as a JSON document to your clipboard.
6. Paste the copied bytecode into a text editor and save it.

Deploy the Smart Contract

In the copied bytecode, the section you need is the `"object"` field. This is the EVM bytecode of a sample smart contract.

```
"object": "6080604052348015610010576000080fd5b506104108061002060000396000f30060
806040526004361061005757576000357c0100000000000000000000000000000000000000000000
000000000000900463
ffffffff1680634ed3885e1461005c5780636d4ce63c146100c5578063d97d663014610155575b
600080fd5b34801561
006857600080fd5b506100c360048036038101908080359060200190820180359060200190808080
```

```
601f01602080910402
6020016040519081016040528093929190818152602001838380828437820191505050505091
929192905050506101
e5565b005b3480156100d157600080fd5b506100da6101ff565b604051808060200182810382 52
838181518152602001
9150805190602001908083836000 5b838110156101 1a57808201518184015260208101905061 00
ff565b505050509050
90810190601f1680156101475780820380516001836020036101000a031916815260200191505b
509250505060405180
910390f35b34801561016157600080fd5b5061016a6102a1565b604051808060200182810382 52
838181518152602001
9150805190602001908083836000 5b838110156101 aa57808201518184015260208101905061 01
8f565b505050509050
90810190601f1680156101d75780820380516001836020036101000a031916815260200191505b
509250505060405180
910390f35b806000908051906020019061 01fb92919061033f565b5050565b6060600080546001
816001161561010002
0316600290048060 1f0160208091040260200160405190810160405280929190818152602001 82
805460018160011615
6101000203166002900480156102975780601f1061026c576101008083540402835291602001 91
610297565b82019190
6000526020600020905b8154815290600101906020018083116102 7a5782900360 1f168201915b
505050505090509056
5b600080546001816001161561010002031660029004806 01f0160208091040260200160405190
810160405280929190
8181526020018280546001816001161561010002031660029004801561033 75780601f1061030c
576101008083540402
835291602001916103375 65b8201919060005260206000020905b8154815290600101906020018 0
831161031a57829003
601f168201915b505050505081565b828054600181600116156101000203166002900490600052
602060002090601f01
6020900481019282601f1061038057805160ff191683800117855561 03ae565b82800160010185
5582156103ae579182
015b828111156103ad57825182559160200191906001019061039256 5b5b5090506103bb919061
03bf565b5090565b61
03e191905b808211156103dd576000816000905550600101610 3c5565b5090565b905600a16562
7a7a72305820a990d4
0b57c66329a32a18e847b3c18d6c911487ffadfed2098e71e8cafa0c980029",
```

In general, the EVM expects two arguments:

- The `to` address.
- The `input` bytecode that's necessary in Ethereum transactions.

To deploy smart contracts, the `to` field is the zero address, and the `input` is the compiled EVM bytecode of the contract. Thus, there are two arguments provided to the `invoke` command. The first one, which was traditionally supposed to be a function name inside the chaincode, is now `0000000000000000000000000000000000000000000000000000000000000000`, and the second argument is the Solidity smart contract bytecode.

1. To deploy the Solidity smart contract on Oracle Blockchain Platform, you can make the following REST proxy call to send the two arguments to the EVM.

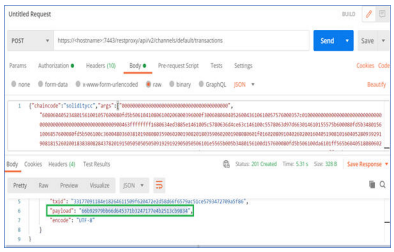
```
{
    "chaincode": "<evmcc-ccid>",
    "args": [
```

```
            "0000000000000000000000000000000000000000",
            "<bytecode-of-the-smart-contract>"
        ],
        "timeout": 0,
        "sync": true
    }
```

The following example uses cURL to deploy the Solidity smart contract to Oracle Blockchain Platform with the name `soliditycc`.

```
curl -L -X POST 'https://<hostname>:443/restproxy/api/v2/channels/
<channelname>/transactions' \
-H 'Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=' \
-H 'Content-Type: application/json' \
--data-raw '{"chaincode":"<evmcc-ccid>","args":
["0000000000000000000000000000000000000000","<bytecode-of-the-smart-
contract>"],"timeout":0,"sync":true}'
```

2. The response `payload` of the transaction is the contract address for your deployed contract. Copy this address and save it. The contract address is used when you run smart contract functions.



In this example, the smart contract address is `66b92979bb66d645371b3247177e4b2513cb9834`.

There are two ways to interact with a deployed smart contract.

1. Interact by using a hash value of the method and input parameters.
2. Interact by using the method name and input parameters directly.

Interacting With the Smart Contract by Using Hash Values

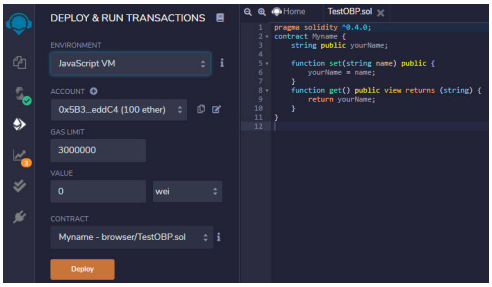
After you have the smart contract address, you can use the following calls to interact with the deployed smart contract via the REST proxy.

To run functions, you use invoke and query transactions but with different parameters. The sample contract contains two functions: `get` and `set`.

In these transactions, the `to` field is the contract address and the `input` field is the function execution hash concatenated with any of the required arguments.

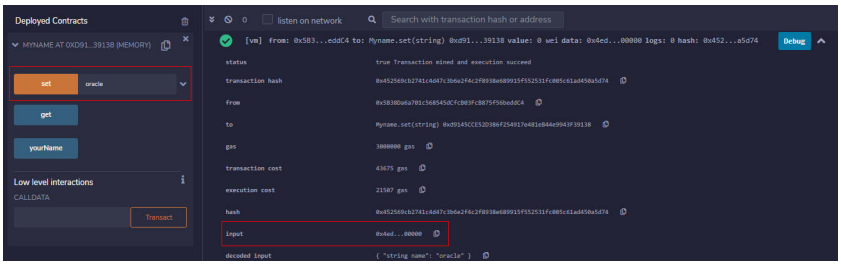
You must acquire the hash of the function execution to run a transaction. A simple way to do this is to run the functions in the Remix IDE and to then copy the hash from the transaction logs:

1. In the Remix IDE, open the **Deploy and Run Transactions** panel. Ensure that your contract is selected in the **Contract** field, and select **Deploy**.



After the deployment completes, the contract will be listed in the **Deployed Contracts** list.

2. Expand the contract in the **Deployed Contracts** list. The smart contract functions are listed.
3. Run a transaction. For the provided example, enter `oracle` and then select **set**.
4. The Terminal window shows the transaction logs. If the transaction logs are minimized, expand them by selecting the log. Copy the value of the `input` field (which is the function execution hash) by selecting the icon next to it. Save this value to the same location as your contract address, removing the leading `0x`.



5. After you have the function execution hash and the contract address, you can run the set transaction on Oracle Blockchain Platform using the hash and address as the raw data arguments.

```
--data-raw '{"chaincode":"<chaincodename>","args":
["<contractaddress>","<setfunctionexecutionhash>"]}'
```

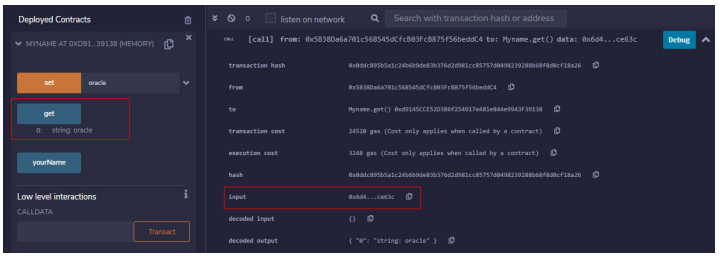
The following example shows how to run the transaction by using cURL:

```
curl -L -X POST 'https://<hostname>:443/restproxy/api/v2/channels/
<channelname>/transactions' \
-H 'Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=' \
-H 'Content-Type: application/json' \
--data-raw '{"chaincode":"solidityc","args":
["66b92979bb66d645371b3247177e4b2513cb9834","4ed3885e000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000066f7261636c650000000000000000000000000000000000
0000000000000000"]}'
```

6. Open the Oracle Blockchain Platform console and check that the transaction is listed in the ledger.

To run another transaction such as a query by using the smart contract's `get` function, you can generate the function execution hash in Remix and then combine it with the contract address:

1. In Remix on the **Deploy and Run Transactions** panel, ensure that your contract is still listed under **Deployed Contracts**. If not, redeploy it.
2. Select **get**. Retrieve and save the input from the transaction as you did with the **set** transaction, removing the leading `0x`.



3. Use this transaction hash and the contract address to run a query transaction against the chaincode deployed on Oracle Blockchain Platform.

```
--data-raw '{"chaincode":"<chaincodename>","args":
["<contractaddress>","<getfunctionexecutionhash>"]}'
```

The following example shows how to run the transaction by using cURL:

```
curl -L -X POST 'https://<hostname>:443/restproxy/api/v2/channels/
<channelname>/chaincode-queries' \
-H 'Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=' \
-H 'Content-Type: application/json' \
--data-raw '{"chaincode":"solidityccc","args":
["66b92979bb66d645371b3247177e4b2513cb9834","6d4ce63c"]}'
```

The returned payload will contain the asset being queried, which in the example case is the string `oracle`.