CODING

First steps with Oracle Cloud Infrastructure SDK for Java

Learn how to control Oracle Cloud Infrastructure resources through Java code.

by Michał Jakóbczyk

April 9, 2021

The cloud often blurs traditional distinctions between infrastructure operations and application development. It is becoming more and more prevalent that developers work not only on application code but also treat cloud infrastructure resources as programmable artifacts. Every major cloud provider, including Oracle Cloud Infrastructure (OCI), offers a set of secure web interfaces to control the lifecycle of cloud resources. These APIs are your gateway to the *cloud control plane*, which is responsible for cloud resource lifecycle management.

As developers, we are mostly interested in designing and running our workloads. Automation is what allows you to fully leverage one of the cloud's key characteristics, namely the rapid self-provisioning of pooled cloud resources. The ability to quickly launch and delete cloud resources such as compute instances, managed Kubernetes clusters, serverless functions, or various data stores, empowers developers to prototype, build applications, and roll out systems faster and in a more efficient way.

How does automation actually work in practice? In this article, you will look closely at the cloud control plane API for OCI.

OCI REST API

Representational State Transfer (REST) architectural style is the most common way for cloud providers to offer their cloud control plane interfaces. Based on that, cloud resources are represented by REST resources. Their lifecycle events can be

easily mapped to HTTP methods (`GET`, `PUT`, `POST`, and `DELETE`
).

Different groups of OCI services use different REST API endpoints depending on the service type and OCI region. For example, to manage data science services in Montreal, you will rely on the `https://datascience.ca-montreal-1.oci.oraclecloud.com` API endpoint. The management of the MySQL Database service in Dubai will require you to use the `https://mysql.me-dubai-1.ocp.oraclecloud.com` API endpoint. (Those URLs won't do anything useful if you click on them in a browser.)

Let me guide you through an example.

To create a new Oracle virtual cloud network (VCN), the `CreateVcn` API is used. The API belongs to the Core Services API. Assuming you want to create this cloud resource in the Zurich region, the following base API endpoint is used: `https://iaas.eu-zurich-1.oraclecloud.com`

To call the `CreateVcn` API, your application sends a `POST` request to the `/20160918/vcns` endpoint. The request payload is formatted as JSON and contains a single element of the `CreateVcnDetails` data type. This data type defines the fields (`"cidrBlock"`, `"dnsLabel"`, and so on) required to properly initialize the newly created VCN. You can see a sample HTTP request in the following code snippet:

```
POST /20160918/vcns
Host: iaas.eu-zurich-1.oraclecloud.com
<authorization and other headers>
{
  "cidrBlock" : "192.168.1.0/24",
  "compartmentId" : "ocid1.compartment.oc1..<
  "displayName" : "sdk-demo-vcn",
  "dnsLabel": "sdkdemo"
}
```

The APIs are intended to deliver the richest set of operations performed on cloud resources. If there is no API resource for a particular operation, such an operation is either not supported at the moment or can be achieved through a sequence of other API calls.

How secure is remote management of cloud resources over these cloud control plane interfaces? Very.

Securing API calls

Communication with a cloud control plane interface is secured during transit by Transport Layer Security (TLS), which provides privacy and integrity for all requests and responses from the OCI

REST API. From a practical point of view, it means your application will interact with HTTPS endpoints only. Unencrypted HTTP traffic is impossible in nearly all cases.

The cloud control plane authenticates and authorizes each request by looking at the *request signature* that is supposed to be delivered as a part of the `Authorization` header. As a result, your application must be able to create this request signature to successfully interact with the OCI REST API. To generate the signature, you need to build a *signing string*, which is composed of parts of the request including, but not limited to, a resource target (such as `/20160918/vcns`) and a hash of the request payload (when present).

To form the signature, the signing string is first encrypted according to the RSA-SHA256 algorithm and then encoded with the BASE64 algorithm. Inside the request's `Authorization` header, together with the request signature (`{request-signature}`), you include a tenancy identifier (`{tenancy-ocid}`), a user identifier (`{user-ocid}`), and a fingerprint (`{key-fingerprint}`) of the public key that corresponds to the private key applied during encryption of the signing string. The public key must be uploaded and assigned to the user in OCI before you send the request. This key is the *API signing key*.

```
Authorization: Signature version="1",
keyId="{tenancy-ocid}/{user-ocid}/{key-finger
algorithm="rsa-sha256", ...,
signature="{request-signature}"
```

Thanks to the request signature, the cloud control plane knows all the pieces required to authenticate, and then authorize, every request.

The good news is that as a Java developer, you do not need to code the request signature creation logic on your own because you can rely on the OCI Software Development Kit (SDK) for Java.

OCI SDK for Java

The OCI SDK for Java is a set of libraries that allows your programs to interact with the OCI REST API in a convenient and efficient way. It provides a broad range of Java classes that are used to make OCI REST API calls and process their results. Additionally, every time the SDK is about to send a request, it also takes the burden of creating a request signature. In this way, you can keep your code simpler.

Look at the following code snippet, which creates the same sample HTTP request shown earlier:

```
// ... authentication-related code
// Creating a client instance to interact wit
VirtualNetworkClient vnClient =  VirtualNetwo
// Creating CreateVcnDetails payload object
CreateVcnDetails details = CreateVcnDetails.b
        .cidrBlock("192.168.1.0/24")
        .compartmentId(compartmentId)
        .displayName("sdk-demo-vcn")
        .dnsLabel("sdkdemo")
        .build();
// Preparing the request object
CreateVcnRequest request = CreateVcnRequest.b
// Calling the CreateVcn REST API
CreateVcnResponse response = vnClient.createV
// Print the OCID of the newly created VCN
if(response.get__httpStatusCode__()==200)
        System.out.println(response.getVcn().
```

First, the code creates a new `VirtualNetworkClient` object using a nested builder class. The builder class's `build()` method requires, as a parameter, an object providing the information used to authenticate requests. More on this object is discussed just a bit later.

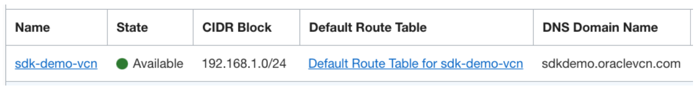
Second, the code prepares a plain old Java object (POJO) that will be eventually serialized into the payload of an API request. In this case, the application will call the `CreateVcn` API. Based on that, the code creates a new `CreateVcnDetails` object and populates its fields accordingly.

The `CreateVcnDetails` class is generated from the corresponding OCI API data type definition. This definition may be useful to choose proper field values.

Third, it creates a new `CreateVcnRequest` object, providing its dedicated builder class with the reference to the `CreateVcnDetails` object.

Finally, the `createVcn` method calls the OCI API.

Figure 1 shows the resulting, newly created VCN cloud resource as seen in the OCI Console.



Name	State	CIDR Block	Default Route Table	DNS Domain Name
sdk-demo-vcn	● Available	192.168.1.0/24	Default Route Table for sdk-demo-vcn	sdkdemo.oraclevcn.com

Figure 1. The VCN cloud resources shown in the OCI Console

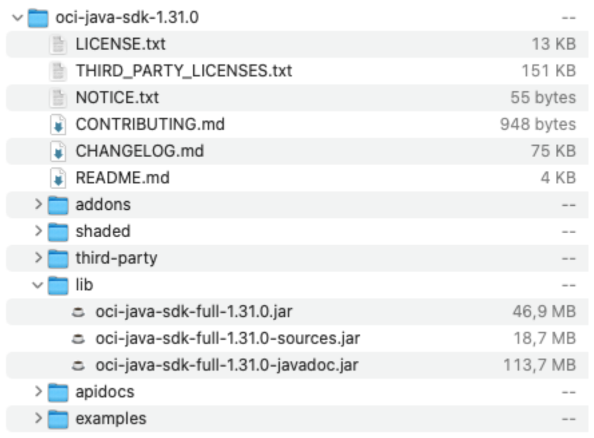
At this stage, you may be asking yourself two fundamental questions:

- Where do I get the OCI SDK for Java classes?
- How do I define the identity of an application in the context of OCI authentication?

Getting the OCI SDK

The SDK code is maintained on GitHub's oci-java-sdk repository, which is also where you can find the latest SDK releases. In that same place, you can browse through existing issues or submit new issues, if needed.

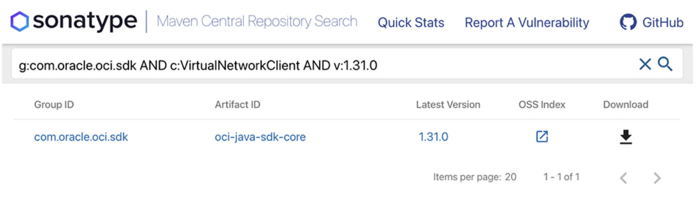
In other words, the most manual way to obtain the official SDK release is to download it from GitHub. The entire file is rather large (around 358 MB for version 1.31.0) because it contains all dependencies, sources, and Javadoc files. **Figure 2** shows the contents.



oci-java-sdk-1.31.0	--
LICENSE.txt	13 KB
THIRD_PARTY_LICENSES.txt	151 KB
NOTICE.txt	55 bytes
CONTRIBUTING.md	948 bytes
CHANGELOG.md	75 KB
README.md	4 KB
addons	--
shaded	--
third-party	--
lib	--
oci-java-sdk-full-1.31.0.jar	46,9 MB
oci-java-sdk-full-1.31.0-sources.jar	18,7 MB
oci-java-sdk-full-1.31.0-javadoc.jar	113,7 MB
apidocs	--
examples	--

Figure 2. The contents of the oci-java-sdk repository

I believe most developers will rather rely on Maven or Gradle to automate the build of Java software. The OCI SDK artifacts (as JAR files) are available in Maven Central under Group ID. You can find the relevant Maven JAR file using the Maven Central Repository Search. **Figure 3** shows how to search for the `VirtualNetworkClient` class (`clause c:VirtualNetworkClient`) in version 1.31.0 (clause v:1.31.0) within the OCI SDK (`clause g:com.oracle.oci.sdk`).



g:com.oracle.oci.sdk AND c:VirtualNetworkClient AND v:1.31.0

Group ID	Artifact ID	Latest Version	OSS Index	Download
com.oracle.oci.sdk	oci-java-sdk-core	1.31.0		

Items per page: 20 1 - 1 of 1

Figure 3. Searching in the Maven repository

Based on the results, you now know all the information required to add the right dependency to your `pom.xml` file.

```
<dependency>
      <groupId>com.oracle.oci.sdk</groupId>
      <artifactId>oci-java-sdk-core</artifact
      <version>1.31.0</version>
    </dependency>
```

You can find the code for a reference implementation in my repository on GitHub. Before running the code, please read the corresponding `README.md` file. Note: You have to be logged in to your GitHub account to access my repository's resources, otherwise you will receive a 404 error.

Now, you need to know where the SDK gets the identity-related information required to compose the request signature for each API call.

Identity of an SDK-based application

Your application must have some kind of *identity*, which will tell OCI which kind of operations are allowed and on which cloud resources it may perform those operations. The information specific to this identity is used to form a complete request signature. The importance of request signatures was described earlier.

The builder of each SDK client class requires a reference (here, `ociAuthProvider`) to an object implementing a subinterface of the `AbstractAuthenticationDetailsProvider` interface.

```
VirtualNetworkClient vnClient =
   VirtualNetworkClient.builder().build(ociAu
```

To form the identity, an application can do one of the following:

- Act on behalf of the named member of an Oracle Identity Cloud Service (IDCS) or an Oracle Cloud Infrastructure Identity and Access Management (IAM) group
- Leverage the *instance principal* mechanism

The privilege to call various OCI APIs is given to *IAM groups*, not to individual IDCS or IAM users. IAM users belong to groups. IDCS users belong to IDCS groups that are mapped to IAM groups. The OCI APIs that can be successfully called by a particular user depend on the user's assignment to one or more IAM groups.

IAM groups are entitled to perform particular actions over selected cloud resource types in a given scope, that is, the entire tenancy or an individual compartment, optionally including its subcompartments as well. This is achieved through the use of *policy statements*.

For example, the following code snippet lists two policy statements that allow users who belong to the group `sandbox-publishers` to upload, download, and remove objects in buckets in the `Sandbox` compartment:

```
allow group sandbox-publishers to read bucket
allow group sandbox-publishers to manage obje
```

The *instance principals* mechanism is both recommended for and, at the same time, useable only for applications running inside OCI on a compute instance, in a Kubernetes pod, or as a serverless function.

Instance principals belong to *dynamic groups*. Matching rules decide on the membership of a cloud resource to a dynamic group. For example, a matching rule can define that all compute instances that are tagged with a specific tag belong to a particular dynamic group. As a consequence, the applications running on these compute instances can successfully call only the OCI APIs that are allowed for this particular dynamic group.

Using instance principals in Java code is pretty straightforward and does not require passing any authentication details on your own.

```
var ociAuthProvider = InstancePrincipalsAuthe
VirtualNetworkClient vnClient =  VirtualNetwo
```

By the way, this article focuses on IAM users. You can find more information on instance principals in the OCI documentation under “Calling services from an instance” or in my book, *Practical Oracle Cloud Infrastructure*.

You can also find information about IDCS users and the mapping of IDCS groups to IAM groups in the OCI documentation under “Managing Oracle Identity Cloud Service users and groups in the Oracle Cloud Infrastructure Console.”

Authentication details for IAM users

An application can act on behalf of a named IAM user. OCI will then let the application use the APIs that are allowed for the IAM group the user belongs to. As a quick recap, OCI authenticates each API request by looking at the *request signature* that is supposed to be delivered as a part of the `Authorization` header. To generate request signatures for each request, an application has to be aware of the following items:

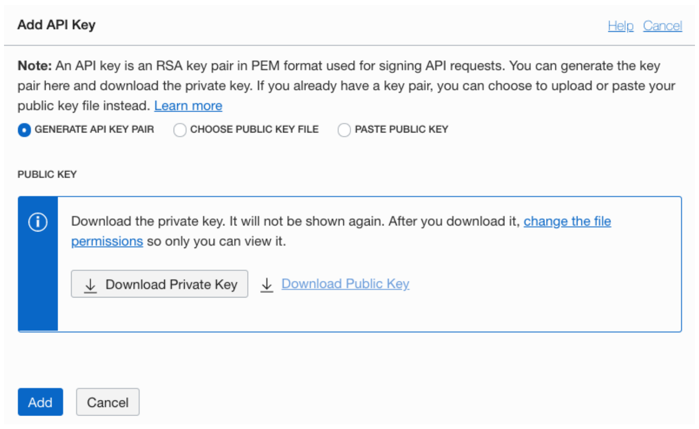
- Tenancy OCID
- OCI region identifier
- IAM user OCID
- API key (previously known as API signing key) in PEM format
    - The public key must be uploaded for the OCI IAM user.
    - The corresponding private key must be readable to the application.
- The fingerprint of the public key
- (Optionally) the passphrase of the private key

If your user is an IDCS user and not an IAM user, do not worry. Everything should still work smoothly. Just make sure the IDCS groups are properly mapped to IAM groups used by policy statements to grant access to OCI APIs.

One way to provide the authentication details is to do it in the code.

```
Supplier<InputStream> privateKeySupplier = ne
var ociAuthProvider = SimpleAuthenticationDet
    .tenantId(tenancyId)
    .region(Region.EU_FRANKFURT_1)
    .userId(userId)
    .privateKeySupplier(privateKeySupplier)
    .passPhrase(privateKeyPassphrase) // if p
    .fingerprint(apiKeyFingerprint)
    .build();
```

Alternatively, you can use an external configuration file for your application. If you have ever used the OCI CLI, you stored the same authentication details in the `.oci/config` file. The configuration file referenced the private key as well. To create such a configuration file, access the OCI Console, go to the relevant `User details` page, click the `API Keys` tab, and click `Add API Key`. A wizard will appear, as shown in **Figure 4**. Make sure you select the `Generate API Key Pair` option, click `Download Private Key`, and then click the `Add` button.


Add API Key

Note: An API key is an RSA key pair in PEM format used for signing API requests. You can generate the key pair here and download the private key. If you already have a key pair, you can choose to upload or paste your public key file instead. Learn more

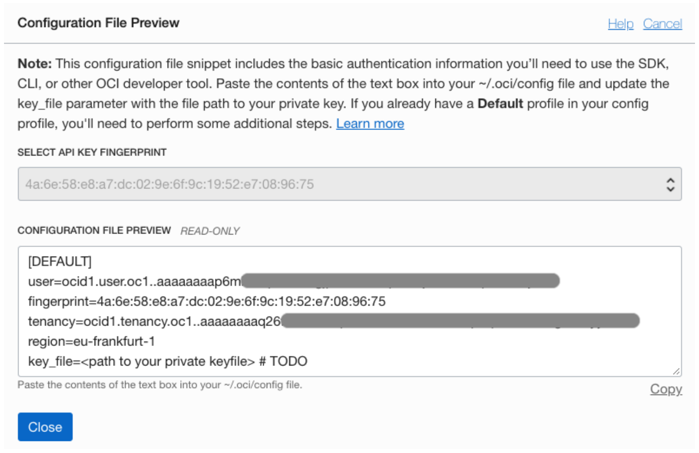
● GENERATE API KEY PAIR ○ CHOOSE PUBLIC KEY FILE ○ PASTE PUBLIC KEY

PUBLIC KEY

Download the private key. It will not be shown again. After you download it, change the file permissions so only you can view it.

Figure 4. Adding the API key

Next, create the `$HOME/.oci/config` file and paste there the displayed details, as shown in **Figure 5**. Do not forget to provide the right path to the newly downloaded private key.



Configuration File Preview

Help Cancel

Note: This configuration file snippet includes the basic authentication information you'll need to use the SDK, CLI, or other OCI developer tool. Paste the contents of the text box into your ~/.oci/config file and update the key_file parameter with the file path to your private key. If you already have a **Default** profile in your config profile, you'll need to perform some additional steps. Learn more

SELECT API KEY FINGERPRINT

4a:6e:58:e8:a7:dc:02:9e:6f:9c:19:52:e7:08:96:75

CONFIGURATION FILE PREVIEW *READ-ONLY*

```
[DEFAULT]
user=ocid1.user.oc1..aaaaaaaap6m[redacted]
fingerprint=4a:6e:58:e8:a7:dc:02:9e:6f:9c:19:52:e7:08:96:75
tenancy=ocid1.tenancy.oc1..aaaaaaaaq26[redacted]
region=eu-frankfurt-1
key_file=<path to your private keyfile> # TODO
```

Paste the contents of the text box into your ~/.oci/config file.

Copy

Close

Figure 5. Creating the $HOME/.oci/config file

To let your application use the authentication details from the `DEFAULT` profile in the `$HOME/.oci/config` file, write code like the following:

```
var configFile = ConfigFileReader.parseDefaul
var ociAuthProvider = new ConfigFileAuthentic
```

Implementing the Claim Check integration pattern

As an integration architect, I could not resist the temptation to implement an integration pattern. So, now you're going to implement the Claim Check integration pattern using

- The OCI SDK for Java
- OCI Object Storage
- OCI Streaming

Architecture. The goal is to implement a loosely coupled and event-driven exchange of large files. The idea is sketched in **Figure 6**.

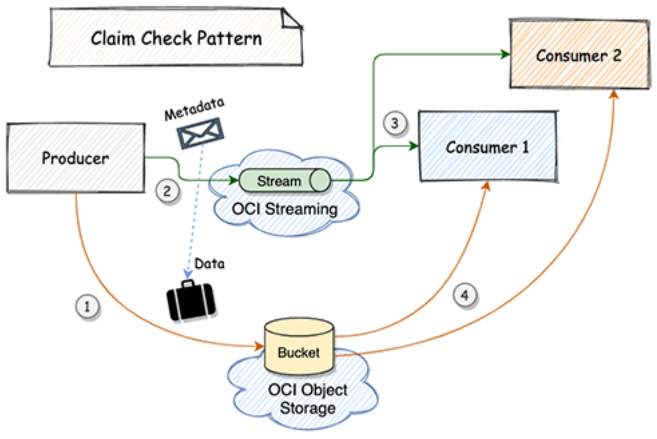


Figure 6. The architecture of the file exchange application

One application (the producer) uploads a large file to OCI Object Storage and publishes the object's metadata (namespace, bucket name, object name) inside a lightweight binary message to OCI Streaming. A second application (the consumer) is periodically retrieving from the stream all newly published messages.

In this way, the consumer is able to decode the object's metadata from the message's payload and download the large file from OCI Object Storage. OCI Streaming is used here to provide event-driven messaging and let the applications exchange the object's metadata quickly using lightweight messages.

The applications are loosely coupled: The producer does not need to establish any synchronous connection to the consumer. It simply uploads the large file to the bucket and sends a message to the stream. This single message can be consumed by multiple applications, allowing all interested parties to collect the large file. I'll call this behavior the publish-subscribe messaging pattern.

Cloud infrastructure. I will assume now that you are using an OCI user (IAM/IDCS) that belongs to an IAM group with administration-level access in some compartment. Please store the Oracle Cloud Identifier (OCID) of this compartment in the `COMPARTMENT_OCID` variable.

To prepare the required cloud infrastructure, you have to create a new OCI Streaming stream and a new OCI Object Storage bucket.

Hint: If you are using Windows, to successfully execute all shell commands from this article, switch to Windows Subsystem for Linux (WSL) or launch a Linux-based virtual machine (or compute instance in the cloud).

Choose some names for these resources and store them as variables.

```
STREAM_NAME=claim-checks
BUCKET_NAME=large-messages
```

Now, execute the following OCI CLI commands:

```
STREAM_OCID=$(oci streaming admin stream crea
BUCKET_OCID=$(oci os bucket create --name $BU
```

The OCID of the stream is stored now in the `STREAM_OCID` variable and the OCID of the bucket is stored in the `BUCKET_OCID` variable. OCI Streaming clients need to know the endpoint of the stream. To obtain the relevant endpoint, execute one more OCI CLI command.

```
STREAM_ENDPOINT=$(oci streaming admin stream
```

You will need all these variables later as the input parameters to the sample application.

The sample application. I have implemented a sample application and published the source on GitHub. You can find the complete codebase here. Here's what the application looks like.

```
ref-oci-sdk-claim-check
├── LICENSE
├── README.md
├── pom.xml
└── src
    └── main
        └── java
            └── io
                └── github
                    └── mtjakobczyk
                        └── javamagazine
                            └── ClaimCheckCli
```

There is a single class called `ClaimCheckClient` that can act either as a producer or as a consumer.

As a producer (`-producer`), the application

- Puts (uploads) the file as an object to an OCI Object Storage bucket
- Creates a message that stores the complete path to the newly created object
- Sends the message to an OCI Streaming stream

As a consumer (`-consumer`), the application polls the OCI Streaming stream for the incoming messages. For each message retrieved from the stream, the consumer

- Decodes the message value (payload) to learn the OCI Object Storage path to an object
- Gets (downloads) the object
- Persists the object as a local file

The `ClaimCheckClient` uses the `DEFAULT` profile in the default OCI configuration file (`$HOME/.oci/config`) to sign your OCI API requests and, in this way, authenticate them. Make sure you have this configuration file available to the application or change the code using one of the alternative authentication providers. For more information, see earlier sections in this article.

Consumers. Start two consumers, each in a separate terminal pane or window, using some of the variables set a few moments ago.

```
java -jar claim-check-client-1.0-jar-with-dep
java -jar claim-check-client-1.0-jar-with-dep
```

The first consumer instance (`receiver-1`) is a member of the `c1` consumer group and saves the downloaded file locally under the following path: `/tmp/receiver-1-large.pdf`. The second consumer instance (`receiver-2`) is a member of the `c2` consumer group and persists the downloaded file as `/tmp/receiver-2-large.pdf`. Running the two consumers in different consumer groups (`c1` and `c2`) will effectively allow both of them to receive the same message.

When the consumers start polling the stream for new messages, you should see something similar to the following in the output:

```
Starting ClaimCheckClient
Using DEFAULT profile from the default OCI co
Preparing OCI API clients (for Object Storage
Querying for Object Storage namespace
Your object storage namespace: yournamespace
ClaimCheckClient acting as consumer
```

Producer. To demonstrate the Claim Check pattern in action, you'll need to create a mock of a large file. In this scenario, the file will have 10 MB and will imitate a PDF file. No, that size is not a joke! From an application messaging perspective, 10 MB is considered a rather large size for a payload. Such a file cannot be processed as a message by OCI Streaming, because it is larger than 1 MB. This is where the Claim Check pattern comes into play: The producer uploads the file to the OCI Object Storage bucket first and subsequently passes the object's path as a message over OCI Streaming to the consumers.

```
# Create a mock of a 10MB PDF
head -c $((10*1024*1024)) /dev/urandom > /tmp
# Run a producer
java -jar claim-check-client-1.0-jar-with-dep
```

You should see something like the following in the output:

```
Starting ClaimCheckClient
Using DEFAULT profile from the default OCI co
Preparing OCI API clients (for Object Storage
Querying for Object Storage namespace
Your object storage namespace: yournamespace
ClaimCheckClient acting as producer
Found file: /tmp/large.pdf (10240kB)
Uploading the file as /n/yournamespace/b/larg
Successfully uploaded the file
Successfully published the message to the str
```

Consumers again. In each individual terminal window, where you started both consumers, you will see new log entries.

```
Consumed message from the stream: /n/jakobczy
Successfully saved file locally as /tmp/recei
Consumed message from the stream: /n/jakobczy
Successfully saved file locally as /tmp/recei
```

Finally, to test whether everything went well, use the good old `diff` utility to compare all three files.

```
diff /tmp/large.pdf /tmp/receiver-1-large.pdf
diff /tmp/large.pdf /tmp/receiver-2-large.pdf
```

The files should be identical.

Implementation details. The application is based on a single class called `ClaimCheckClient`. This class can act either as a producer or as a consumer.

A quick look at the Maven `pom.xml` file shows that you compiled the source with Java 15 language features (which also means you need to have at least Java 15 on your build machine) and you rely on the following dependencies:

- `picocli`
- `oci-java-sdk-core`
- `oci-java-sdk-objectstorage`
- `oci-java-sdk-streaming`
- `slf4j-api with slf4j-simple`

Now, check the Java code. Open the ClaimCheckClient.java file in another browser tab.

I have chosen the picocli framework to implement a neat CLI with convenient features such as exclusive argument groups (different parameters for `-consumer` and `-producer` modes) and a single-line program start.

```
new CommandLine(new ClaimCheckClient()).execu
```

The `ClaimCheckClient` implements the `Callable<Integer>` interface. The core application logic is implemented in the `call()` method. Picocli calls this method after parsing and validating the input parameters.

Look at the behavioral Unified Modeling Language (UML) sequence diagram in **Figure 7**.

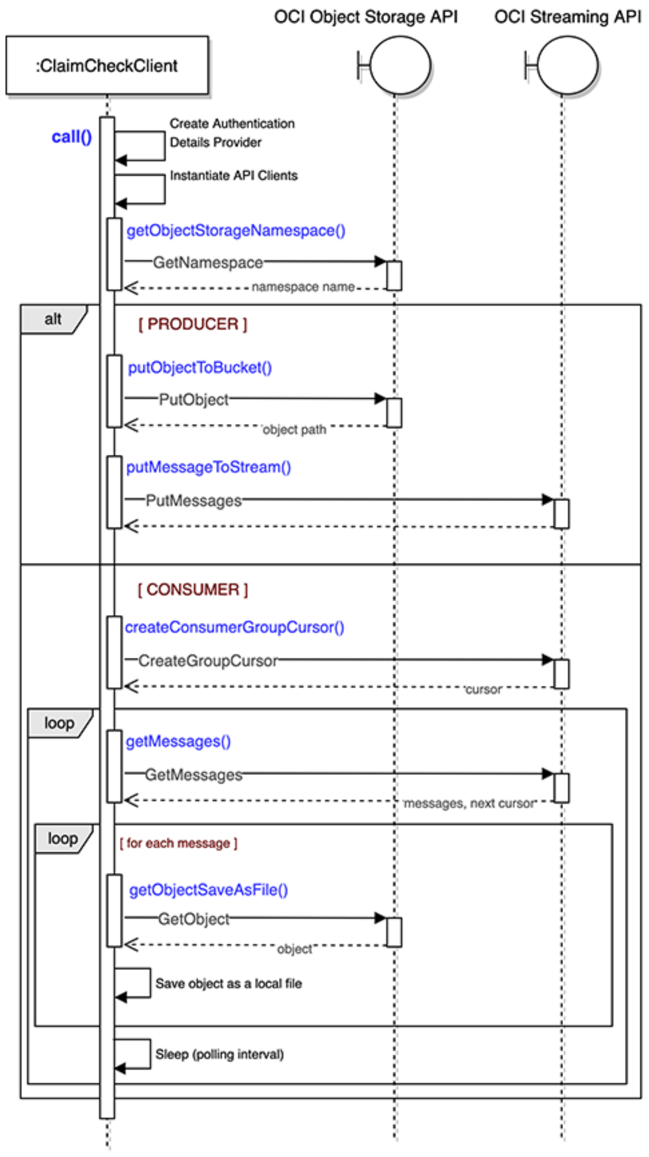


Figure 7. The UML sequence diagram for the ClaimCheck application

The diagram shows the sequence of steps, including calls to OCI APIs. As shown in the diagram, the core application logic consists of four stages.

- Create authentication details provider instance (`ociAuthProvider`).
- Instantiate the OCI API clients (`osClient` and `streamClient`).
- Call the OCI Object Storage API to get the OCI Object Storage namespace (`osNamespace`).
- Execute consumer or producer logic, depending on the input parameters.

```
var configFile = ConfigFileReader.parseDefaul
var ociAuthProvider = new ConfigFileAuthentic

var osClient = ObjectStorageClient.builder().
var streamClient = StreamClient.builder().end
```

```
    var osNamespace = getObjectStorageNamespace(o

    if (clientModeArgs.consumerArgs != null && cl
        // CONSUMER MODE LOGIC
    }
    if (clientModeArgs.producerArgs != null && cl
        // PRODUCER MODE LOGIC
    }
```

The `getObjectStorageNamespace` method calls the OCI Object Storage API to get the name of the tenant's OCI Object Storage namespace. Each OCI tenant is assigned a unique and constant OCI Object Storage namespace. Knowing the namespace name is necessary to upload and download files using the OCI Object Storage API. A dedicated builder class takes the OCID of the compartment and builds an instance of the `GetNamespaceRequest`.

The OCI API call is done by the `ObjectStorageClient` object (`osClient`).

The `getNamespace(GetNamespaceRequest)` method returns the result of the OCI API call in a form of a `GetNamespaceResponse` object. This object includes an HTTP status code. You can use the code to find out what really happened in OCI. If the operation was successful (`HTTP 200`, in this case), extract the name of the OCI Object Storage namespace and return it to the `call()` method.

```
    private String getObjectStorageNamespace(Obje
        logger.info("Querying for Object Storage 
        // Get Object Storage Namespace
        // https://docs.oracle.com/en-us/iaas/api
        var getNamespaceRequest = GetNamespaceReq
                                        .compartmentI
                                        .build();
        var getNamespaceResponse = osClient.getNa
        var getNamespaceResponseCode = getNamespa
        if(getNamespaceResponseCode!=200) {
            logger.error("GetNamespace failed - H
            System.exit(1);
        }
        var osNamespace = getNamespaceResponse.ge
        logger.info("Your object storage namespac
        return osNamespace;
    }
```

The producer's logic calls the `putObjectToBucket` method to upload the large file to a bucket in OCI Object Storage. The method returns a path to the newly uploaded object (for example, `/n/yournamespace/b/large-files/o/large.pdf`). Next, the path is passed to the `putMessageToStream` method, which sends the path encapsulated in an OCI Streaming message. Each of the two methods performs a single OCI API call.

```
// PRODUCER MODE
if (clientModeArgs.producerArgs != null && cl
    var filePath = Paths.get(filePathStr);
    if (Files.exists(filePath) && Files.isReg
        logger.info("Found file: {} ({} kB)",
        var osPathToObject = putObjectToBucke
        putMessageToStream(streamClient, osPa
    } else {
        logger.error("{} is not a regular fil
        return 1;
    }
}
```

The consumer’s logic calls the `createConsumerGroupCursor` method, which creates a cursor used to consume the stream. Inside the method code, notice the `Type.Latest` cursor type, effectively polling only for new messages and ignoring those that are already present in the stream. Next, the logic enters an infinite loop to poll the OCI Streaming API every few seconds.

In each iteration, the `getMessages` method retrieves up to 10 messages and returns them with the next cursor. For each message, the code calls the `getObjectSaveAsFile` method to decode the message payload as the path to the object, download the object from OCI Object Storage, and save it locally as a file.

```
// CONSUMER MODE
if (clientModeArgs.consumerArgs != null && cl
    String cursor = createConsumerGroupCursor
    do {
        var getMessagesResponse = getMessages
        String nextCursor = getMessagesRespon
        List<Message> messages = getMessagesR
        for (Message msg : messages) {
            String osPathToObject = new Strin
            logger.info("Consumed message fro
            getObjectSaveAsFile(osClient, osP
        }
        cursor = nextCursor;
        Thread.sleep(clientModeArgs.consumerA
    } while (true);
}
```

Looking closer at the structure of all these methods that use OCI SDK classes inside, notice that the sequence of steps to call the OCI API from Java code is the same.

- Create a request object (for example, of the `GetObjectRequest` class) to store OCI API request parameters.
- Pass this request object to the relevant method of the OCI API client class (`osClient.getObject(getObjectRequest)`).
- Check the HTTP status code (`getObjectResponse.get__httpStatusCode__()`).

- Optionally, extract the data from the response object (`getObjectResponse.getInputStream()`).

Conclusion

The OCI SDK is a set of Java libraries that allows your programs to interact with the OCI REST API in a convenient and efficient way. The SDK code is maintained on GitHub. OCI SDK artifacts (or, simply, JAR files) are available in Maven Central under Group ID `com.oracle.oci.sdk`.

OCI authenticates each API request by looking at the request signature that is supposed to be delivered as a part of the `Authorization` header. An application can act on behalf of a named IAM/IDCS user. OCI will then let the application use the APIs that are allowed for the IAM group the user belongs to. Alternatively, an application running inside OCI (on a compute instance, in a Kubernetes pod, or as a serverless function) can rely on the instance principals mechanism.

In this article, I have guided you in taking first steps for the OCI SDK for Java, and I described a complete solution that implements the Claim Check pattern with OCI Object Storage and OCI Streaming.

Dig deeper

- OCI REST APIs
- OCI SDK for Java
- Core Services API
- Getting started with Kubernetes
- Hello, Coherence Community Edition, Part 3: Packaging, deployment, scaling, persistence, and operations with Java
- OCI SDK for Java Cloud Shell quick start

Michał Jakóbczyk

Michał Jakóbczyk is a cloud integration architect based in Europe and the author of *Practical Oracle Cloud Infrastructure: Infrastructure as a Service, Autonomous Database, Managed Kubernetes, and Serverless* (Apress, 2020). He consults with and provides advice to clients on integration architecture and cloud infrastructure. He holds a bachelor of science in engineering in the field of decision support systems and computer science from Warsaw University of Technology.