Case Study on Building Data-Centric Microservices

Part I - Getting Started

May 26, 2020 | Version 1.0

DISCLAIMER

TABLE OF CONTENTS

INTRODUCTION

From mom-and-pop brick and mortar to behemoth merchants, modern applications are based on Microservices architecture. Such an architecture brings undeniable benefits including: agile development, testing and deployment of independents services (DevOps, CI/CD), polyglot programming, polyglot persistence (data models), fine-grained scalability, higher fault tolerance, Cloud scale tracing, diagnosability, and application monitoring, and so on. However, such a Microservice architecture comes with inherent challenges including increased communication, data consistency, managing transactions across services, and overall increased work. There are periodic debates about the granularity of Microservices (micro or macro?) but the consensus is that the benefits outweigh the challenges. These challenges are usually addressed at design level in splicing into a full application bounded domains of composable modular subservices. Modular principles have overall benefits, which when implemented well, hold the keys to success with Microservices.

The goal of this paper (first of a multi-part series) is to describe in detail how to build and deploy a data-centric application, and how the Cloud services on the Oracle Cloud Infrastructure along with the Oracle Autonomous Database help simplify the key implementation challenges. While we illustrate the key design principles on the Oracle Cloud, the underlying architecture and open interfaces can be implemented and deployed on-premises to get started.

ARCHITECTURE OVERVIEW

This section discusses the requirements and challenges for a Microservices-based application including: polyglot persistence, persisting events and state, and data consistency across Microservices.

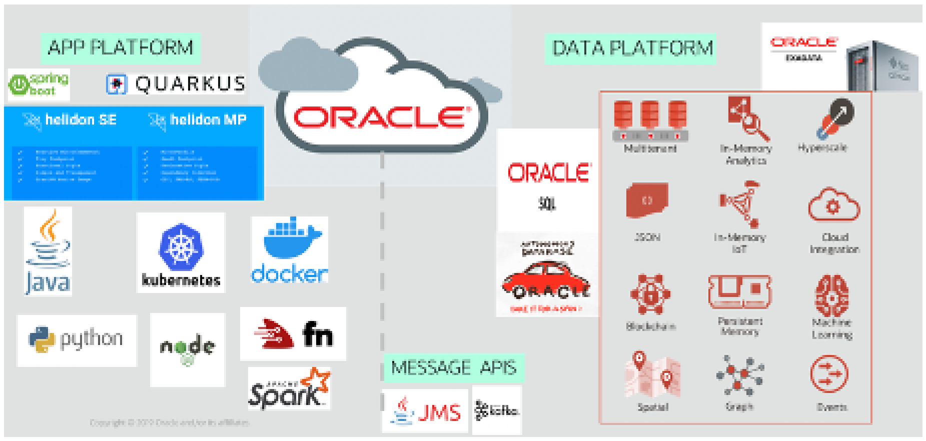
Before You Begin

We highly recommend to understand: (i) the Twelve Factor App methodology; and (ii) the Cloud Native Computing Foundation frameworks and technologies as your Microservices platform is based on it. A simple overview of the twelve factor app is in the table below as these are good practices and we highlight some of these in the simplified best practices to build microservices with the Oracle Converged Database.

FACTOR	IMPLICATIONS
Code base	Track revisions in one codebase, with potentially many deployments
Dependencies	Explicitly declare and isolate dependencies
Config	Store config in the environment
Backing Services	Treat backing services as attached resources
Build, release, run	Strictly separate build and run stages
Processes	Execute the app as one or more stateless processes
Port binding	Export services via port binding
Concurrency	Scale out via the process model
Disposability	Maximize robustness with fast startup and graceful shutdown
Dev/Prod Parity	Keep development, staging, and production as similar as possible
Logs	Treat logs as event streams
Admin Processes	Run admin/management tasks as one-off processes

A typical Microservice platform such as the Oracle Cloud Infrastructure (OCI) is made up of: an Image Registry (OCIR), Microservice development frameworks (such as Helidon), a Container (Docker or CRI-O), an Orchestrator (the OCI Kubernetes Engine a.k.a. OKE), a Service broker (Open Service Broker) for provisioning Cloud services, an API gateway, an Events service (OCI Events Service), a Service Mesh (such as Istio) and service graph visualization tools (such as Kiali), a telemetry and tracing framework (opentracing with Jaeger or equivalent), metrics and an analytic and alerting dashboard (Grafana or equivalent), storage services (OCI Object Store) and Oracle Converged Database with built-in Transactional Event Queues (TEQ) (such as the Autonomous Database or Exadata Cloud Service). Additional services like Oracle Functions, Oracle Machine Learning and Spark (OCI Dataflow), etc. can add real-time AI/ML capabilities to Microservices to make them intelligent.

On the data side, the converged database supports Document (JSON), Spatial, Graph, and Relational data in a Container database (CDB) with one or more pluggable databases (PDB). This paradigm nicely allows for deploying SaaS services with multi-tenancy and microservices with data isolation to separate PDBs when needed.



Our Canonical Application

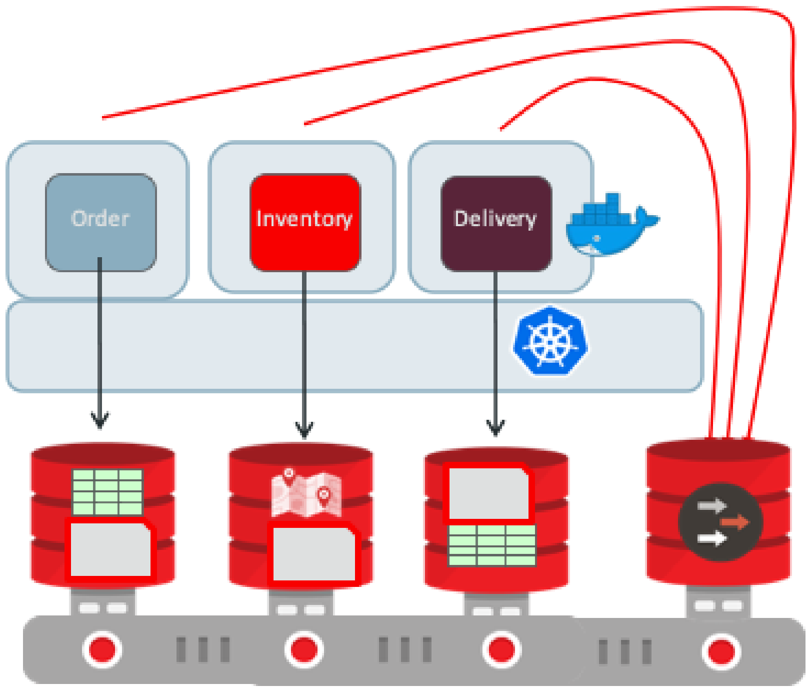
This data-centric microservices application performs a transaction -- i.e., booking a mobile food order (e.g. appetizer, main course, dessert) from one or many restaurants, wherein an order may fail if an item is not available in inventory, or delivery person is not available to deliver to the destination. The Oracle Learning Library has a set of Hands-on-Labs that provides a self-service walkthrough of creating a Microservices deployment on the Oracle Cloud.

As pictured hereafter, each service:

- Performs a specific task: book or reserve either a mobile food order, check for inventory, and delivery of the item
- Is implemented using either Java, Python or Node.js
- Communicates with other services via Events using Oracle Transactional Event Queues (TEQ) transactional messaging
- Accesses both relational and NoSQL (JSON) data models in a dedicated Pluggable database

The key challenges include: adopting a polyglot persistence strategy, ensuring the atomicity of persisting data and events, ensuring data consistency, the handling of transactions across microservices and compensation logic when failures are encountered.

Microservices Application Architecture



Converged Database with Transactional Event Queues

Polyglot Persistence – Converged Database

Data-driven applications use multiple data models including Relational, Document/JSON, Graph, Spatial, IoT, and Blockchain data. Your challenge consists in selecting either a specialized engine for each data model or a multi-model engine. Does your use case require integrating or querying across data models? For example, how would you persist a rich catalog containing text, images, graphs and documents using a single data model? Are you prepared for pulling data over here then injecting over there? Are you prepared for heterogenous administration and patching mechanisms or looking for a unique/homogeneous administration mechanism across data models.

The Oracle Converged Database is a multi-model engine with pluggable databases which simplifies the polyglot persistence challenge. This architecture brings the best of both worlds; it allows you to either specialize a pluggable database into a specific data model store or turn a pluggable database into a general purpose multi-model store. In our canonical application, each Microservice uses a dedicated pluggable database for storing both Relational and Document/JSON data. It also uses the in-built messaging and streaming layer (Transactional Event Queues) which provide transactional messaging to simplify application coding.

Persisting Events And State – Transactional Event Queues

In this application, we opt for event-driven communication between microservices often referred to as event sourcing. This is asynchronous communication with an event broker and an event store, which serves as the sole source of truth. Each microservice gets notified when an event, in which it has expressed interest via subscription, is produced. Upon completing its task, the service must persist changes to data and persist an event for notifying other services. The challenge consists in persisting the state and the event, atomically, i.e. in the same local transaction (XA being an anti-pattern in microservices architecture). This asynchronous communication alleviates issues related to microservice environments such as network outages and changes, transient availability of services themselves, back pressure, need for retry logic, etc. It also allows for loose coupling with other services which facilitates independent continuous development/test/deployment and fine-grained scaling.

Several techniques or frameworks are used to address such challenges, notably: the Outbox pattern, using a table as a message queue, and the Transactional Event Queue.

We pick Oracle Transactional Event Queues (TEQ) for the transactional messaging it provides, as it is built into the Oracle Database. TEQ's APIs include JMS, Java, PL/SQL, C, and Kafka Java client. TEQ serves the purpose of an event store and event broker, where each microservice maintains an incoming event queue to subscribe to other Microservices' outgoing queues, and an outgoing queue to examine the incoming event, process it, and then respond. This is in contrast to using an API gateway, which also can be used in the context of Microservices, especially if there are very few microservices and the overhead of maintaining API changes is not overwhelming. Using an event store is more scalable since messages are asynchronous. TEQ delivers messages exactly once, by following transaction semantics in the database (not available for the Kafka Java Client API), and by propagating messages/events between queues in separate databases. The producers and subscribers of events can send and consume messages within the exactly-once transaction semantics even in the presence of failures and retries. This is one of the key advantages of the simplification of the messaging flows with TEQ in an Event-driven architecture.

Transactions Across Microservices – The Saga Pattern

In our canonical application, each service persists data in its dedicated local database. However, until the business transaction is completed or rolled back entirely, the data is not consistent at any given time but rather will eventually become consistent. As an example, the committed changes to the available orders made by the order service of the current mobile ordering are not permanent as the order might be rolled back; the value of the available inventory read by another concurrent transaction is not consistent at the time of the reading as it will be changed when-if the current business transaction is rolled back. How do you ensure eventual consistency in long running transactions?

In an asynchronous communication system, the two-phase commit protocol with distributed locking is a no-go. The Saga pattern helps ensure the consistency of long-running business transactions. An application could use the orchestration Saga pattern with a Frontend service as the coordinator -- as opposed to the choreography Saga where services communicate and coordinate among themselves without a coordinator. However, the Saga pattern comes with the following challenges/issues:

1. It does not guarantee Isolation (as in A.C.I.D.); in the order booking example, the amount of available inventory is not the same upon repeatable reads.
2. High development/test costs of the compensating changes made by local transactions in the event of failure of the entire business transaction; the compensation code may amount for up to 80% of the microservices code which adds to huge code maintenance overhead.
3. And there is additional complexity due to human interaction

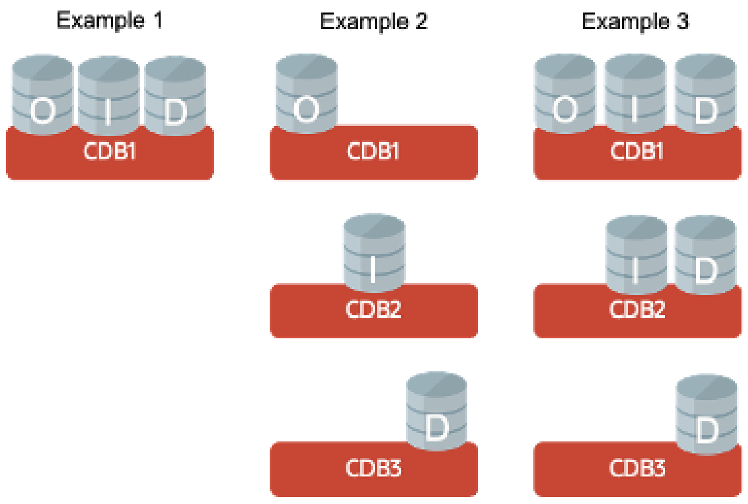
Saga Support could be entirely programmatic (DIY) or implemented via frameworks such as the Long Running Activity for Microprofile API, or via database primitives. The Saga pattern for Microservices will be the subject of a future whitepaper in this series.

Data Distribution Across Microservices

One of the best practices is to design microservices within bounded contexts, thereby guaranteeing data independence and avoiding transactions across microservices as much as possible. The additional benefits of data independence are: agility, as each service can be implemented by a small team with no dependency on others; and the freedom for schema or database reorganization and placement. In principle, each microservice comes with its dedicated database; however, such principle must not be taken to the letter. The following table discusses some of these possible choice.

MICROSERVICE DATABASE ISOLATION	BENEFITS	CONSIDERATIONS /ISSUES	ANALYTICS /REPORTING
Table(s)/View(s) per microservice	Simplest initial maintenance	Role-based access control Availability, maintenance restrictions as all use same database	Views
Schema per microservice	Simple initial maintenance	Availability, maintenance restrictions as all use same database	Cross schema queries/views
PDB per microservice	Ultimate flexibility (PDBs can be deployed together or separately, and this can be easily changed over time)	Combine or separate based on availability and maintenance needs	Cross PDB queries/views for PDBs in same CDB Database links for PDBs in separate CDBs
Non-Multitenant Database per microservice	Highest isolation	Separate availability, maintenance Extra steps required to consolidate	Database Links

The following figure depicts some of the many possible deployment or placement options. Three Oracle pluggable databases (PDBs) associated with three notional microservices: Order (O), Inventory (I), and Delivery (D). In the first example, all three PDBs are deployed within the same Container database (CDB) and would be maintained together with the same database release, patch and availability levels. In the second example, the PDBs are deployed each on its own CDB; each potentially with its own database release, patch level, and availability. The third example depicts how PDB sharding can be used to provide additional scalability by sharding Delivery three ways, and sharding Inventory two ways, while Orders remains unsharded.

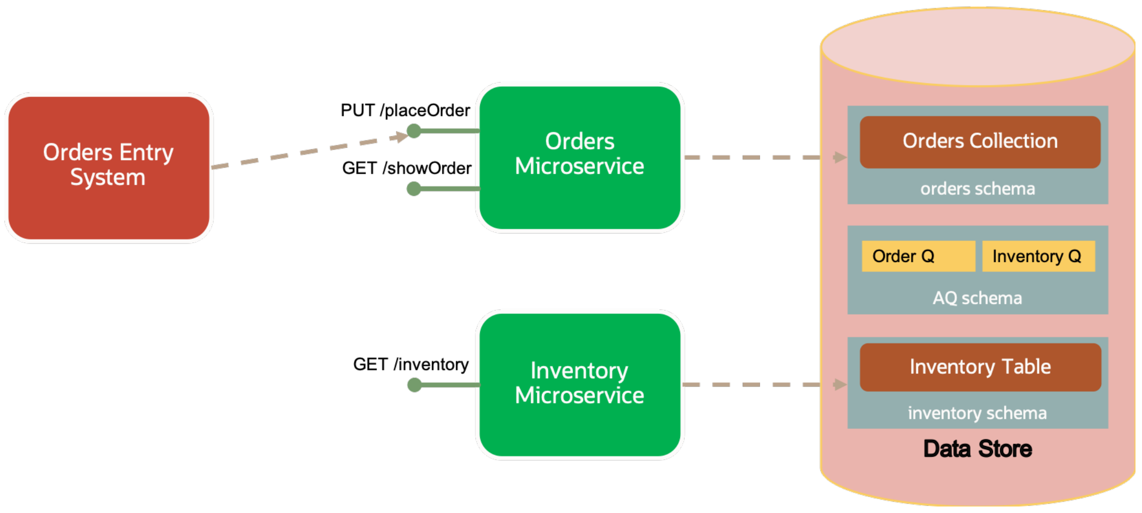


CASE STUDIES

To compare and contrast microservice implementations, we defined a standard specification, test environment, and test harness. The specification is a subset of the Mobile Food Order application described earlier in this paper.

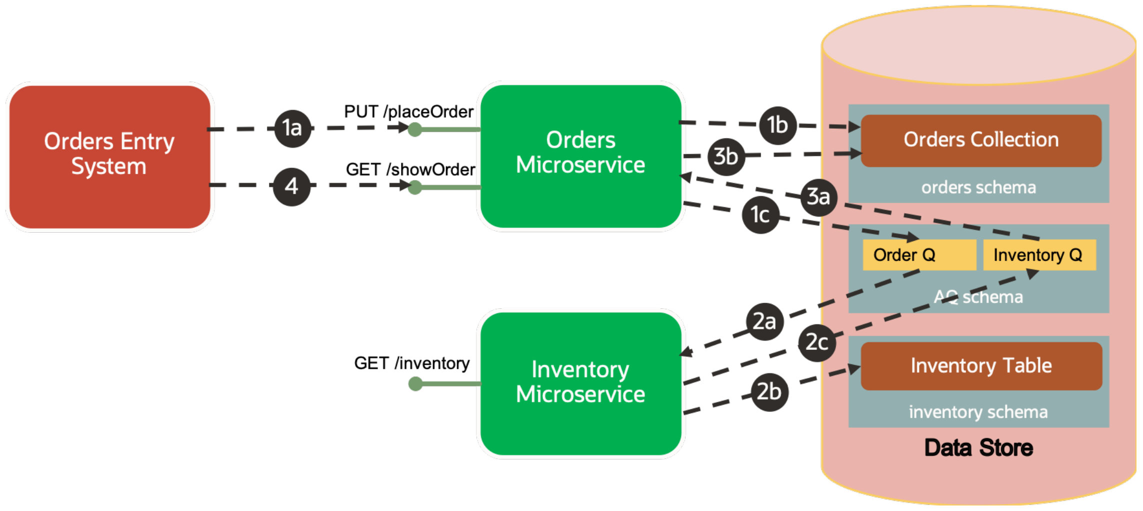
Microservice Specification

The specification describes two microservices, Orders and Inventory, working together to place and fulfill orders. The microservices function independently with their own database schemas. The Orders microservice persists orders as JSON documents in a SODA collection. The Inventory microservice persists inventory information using a relational table. The services interact with each other through Oracle AQ messaging. The microservices respond to HTTP REST requests and to messages sent over AQ messaging.



MESSAGE FLOW

The diagram below shows how an order is placed and how messages flow between the two microservices to check for and allocate inventory.



1. Place Order
    a. An external application calls the PUT /placeOrder interface on the Orders microservice with the order details.
    b. The order is inserted into the orders collection.
    c. A message is placed on the orderqueue queue.
    d. The insertion of the order and enqueuing of the message are committed to the database as a single transaction.
2. Inventory Check
    a. The Inventory microservice receives the message on the orderqueue queue.
    b. The inventory is checked and allocated
    c. A message with the inventory details is sent on the inventoryqueue queue.
    d. The receipt of the message, allocation of inventory, and sending of the message are committed to the database as a single transaction.
3. Order Update
    a. The Orders microservice receives the message on the inventoryqueue queue
    b. The order document is updated in the orders collection.
    c. The dequeuing of the message and update of the order document are committed to the database as a single transaction.
4. At any time, the order status can be queried through the GET /showOrder interface on the Orders microservice.

FUNCTIONALITY

MICROSERVICE	OPERATION	SPECIFICATION
Orders	PUT /placeOrder	• Insert order (JSON) into the “orders” collection using the SODA interface • Post a message to the “orderqueue” using the AQ interface • Commit • Return the order (JSON)
	GET /showOrder	• Retrieve the order by orderid from the “orders” collection using the SODA interface • Return the order (JSON)
	Inventory Queue Consumer	• Retrieve a message from the “inventoryqueue” using the AQ interface • Update the order with the inventory status • Commit
Inventory	GET /inventory	• Query an inventory row from the inventory table by inventory_id using the relational SQL interface • Return the inventory information (JSON)
	Order Queue Consumer	• Retrieve a message from the “orderqueue” using the AQ interface • Check and decrement the inventory in the inventory table using the relational SQL interface • Post a message to the “inventoryqueue” with the inventory status using the AQ interface • Commit

MESSAGE FORMAT

QUEUE	OPERATION	JSON FORMAT (BY EXAMPLE)
orderqueue	PUT /placeOrder	`{"orderid": “000012”, "itemid": “34”, "deliverylocation": “London”, "status": “pending” }`
inventoryqueue	Order Queue Consumer	`{“orderid”: “000012”, “action”: "inventoryexists", “inventorylocation”: “New York”}`

MICROSERVICE CONFIGURATION

PARAMETER	FUNCTION
DB_CONNECT_STRING	Database TNS connect string.
DB_USER	Database username.
DB_PASSWORD	Database login password. Could be different on each database deployment and may vary over time.
DB_CONNECTION_COUNT	Number of database connections in the connection pool for each worker process.
WORKERS	Optional. Number of worker processes to be deployed per container.
HTTP_THREADS	Optional. Number of HTTP threads to be deployed per worker process.
DEBUG_MODE	1 – enable debugging, 0 – disable debugging. Used when testing and debugging the application.
AQ_CONSUMER_THREADS	Number of threads consuming AQ messages.
QUEUE_OWNER	Database schema that owns the database queues.

DATABASE ACCESS

The microservices are implemented with connection pooling with Oracle FAN enabled.

DATABASE SCHEMAS

SCHEMA	DEFINITION

orders	`    l_metadata := '{`
	`        "keyColumn":{`
	`            "assignmentMethod": "CLIENT", "name": "ORDERID", "sqlType": "VARCHAR2"`
	`        }`
	`    }';`
	
	`    collection := DBMS_SODA.create_collection('orders', l_metadata);`

inventory	`create table inventory (`
	`  inventoryid varchar(16) primary key,`
	`  inventorylocation varchar(32),`
	`  inventorycount integer);`
	
	`insert into inventory values('24','New York', 10000);`
	`insert into inventory values('30','New York', 10000);`
	`insert into inventory values('31','New York', 10000);`
	`insert into inventory values('32','New York', 10000);`
	`insert into inventory values('33','New York', 0);`
	`insert into inventory values('34','New York', 0);`

The Orders microservice is given:

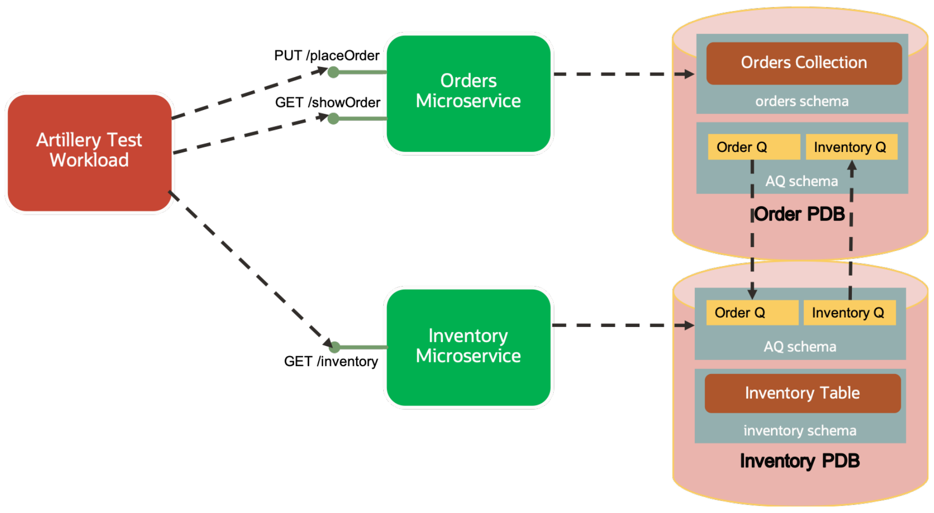
- Login access to the ORDERS schema containing the SODA “orders” collection.
- Enqueue access to the “orderqueue” and dequeue access to the “inventoryqueue” in the AQ schema.

The Inventory microservice is given:

- Login access to the INVENTORY schema containing the inventory table.
- Enqueue access to the “inventoryqueue” and dequeue access to the “orderqueue” in the AQ schema.

The fine grained security roles and privileges that are available with Oracle are useful for controlling the behavior of Microservices.

Test Environment Specification



DATABASE DEPLOYMENT

The ORDERS and INVENTORY schemas deployed on separate pluggable databases (PDBs) within the same container database (CDB). All case studies shared the same two node RAC release 19.6 database deployed on Oracle Cloud Infrastructure.

Authentication was by username and password.

The database connection string had the following structure:

```
(DESCRIPTION=(CONNECT_TIMEOUT=5)(TRANSPORT_CONNECT_TIMEOUT=3)(RETRY_COUNT=3)(RETRY_DELAY=3) (ADDRESS_LIST=
(LOAD_BALANCE=on)(ADDRESS=(PROTOCOL=TCP)(HOST=<SCAN ADDRESS>)(PORT=1521)))(CONNECT_DATA=
(SERVICE_NAME=<DATABASE SERVICE NAME>)))
```

Access to the database was via a database service. The database service was created as follows:

```
srvctl add service -db <DB NAME> -service <SERVICE NAME> -preferred "DBRAC1,DBRAC2" -pdb <PDB NAME> -
notification TRUE
```

AQ CONFIGURATION

Each PDB contains an AQ schema with orders and inventory queues. Messages propagated between the queues in the schemas in each PDBs. The following table details the queue and propagation configuration:

SCHEMA	DEFINITION

AQ (Orders PDB)	`DBMS_AQADM.CREATE_QUEUE_TABLE (`
	`queue_table          => 'ORDERQUEUE',`
	`queue_payload_type   => 'RAW',`
	`multiple_consumers   => true);`
	
	`DBMS_AQADM.CREATE_QUEUE (`
	`queue_name          => 'ORDERQUEUE',`
	`queue_table         => 'ORDERQUEUE');`

```
DBMS_AQADM.grant_queue_privilege (
   privilege     =>     'ENQUEUE',
   queue_name    =>     'ORDERQUEUE',
   grantee       =>     'orders',
   grant_option  =>      FALSE);

DBMS_AQADM.START_QUEUE (
queue_name          => 'ORDERQUEUE');

DBMS_AQADM.CREATE_QUEUE_TABLE (
queue_table          => 'INVENTORYQUEUE',
queue_payload_type   => 'RAW');

DBMS_AQADM.CREATE_QUEUE (
queue_name          => 'INVENTORYQUEUE',
queue_table         => 'INVENTORYQUEUE');

DBMS_AQADM.grant_queue_privilege (
   privilege     =>     'DEQUEUE',
   queue_name    =>     'INVENTORYQUEUE',
   grantee       =>     'orders',
   grant_option  =>      FALSE);

DBMS_AQADM.START_QUEUE (
queue_name          => 'INVENTORYQUEUE');

create database link INVENTORY.<GLOBAL_NAME> connect to aq identified by
"<PASSWORD>" using 'INVENTORY';

DBMS_AQADM.add_subscriber(
   queue_name=>'aq.ORDERQUEUE',
   subscriber=>sys.aq$_agent(null,'aq.ORDERQUEUE@INVENTORY.<GLOBAL_NAME>',0),
   queue_to_queue => true);

dbms_aqadm.schedule_propagation
      (queue_name        => 'aq.ORDERQUEUE'
      ,destination_queue => 'aq.ORDERQUEUE'
      ,destination       => 'INVENTORY.<GLOBAL_NAME>'
      ,start_time        => sysdate --immediately
      ,duration          => null    --until stopped
      ,latency           => 0);     --No gap before propagating
```

AQ (inventory PDB)

```
DBMS_AQADM.CREATE_QUEUE_TABLE (
queue_table          => 'ORDERQUEUE',
queue_payload_type   => 'RAW');

DBMS_AQADM.CREATE_QUEUE (
queue_name          => 'ORDERQUEUE',
queue_table         => 'ORDERQUEUE');

DBMS_AQADM.grant_queue_privilege (
   privilege     =>     'DEQUEUE',
   queue_name    =>     'ORDERQUEUE',
   grantee       =>     'inventory',
   grant_option  =>      FALSE);

DBMS_AQADM.START_QUEUE (
queue_name          => 'ORDERQUEUE');

DBMS_AQADM.CREATE_QUEUE_TABLE (
queue_table          => 'INVENTORYQUEUE',
queue_payload_type   => 'RAW',
multiple_consumers   => true);

DBMS_AQADM.CREATE_QUEUE (
queue_name          => 'INVENTORYQUEUE',
queue_table         => 'INVENTORYQUEUE');

DBMS_AQADM.grant_queue_privilege (
   privilege     =>     'ENQUEUE',
   queue_name    =>     'INVENTORYQUEUE',
   grantee       =>     'inventory',
   grant_option  =>      FALSE);

DBMS_AQADM.START_QUEUE (
queue_name          => 'INVENTORYQUEUE');
```

```
create database link ORDERS.<GLOBAL_NAME> connect to aq identified by
"<PASSWORD>" using 'ORDERS';


DBMS_AQADM.add_subscriber(
   queue_name=>'_aq.INVENTORYQUEUE',
   subscriber=>sys.aq$_agent(null,'${LANG}_aq.INVENTORYQUEUE@<GLOBAL_NAME>',0),
   queue_to_queue => true);
END;


dbms_aqadm.schedule_propagation
      (queue_name        => 'aq.INVENTORYQUEUE'
      ,destination_queue => 'aq.INVENTORYQUEUE'
      ,destination       => 'ORDERS.<GLOBAL_NAME>'
      ,start_time        => sysdate --immediately
      ,duration          => null    --until stopped
      ,latency           => 0);     --No gap before propagating
```

MICROSERVICE DEPLOYMENT

We containerized each of the microservices using Docker and deployed each with two replicas on Oracle Container Engine for Kubernetes fronted by a load balancer.
To deploy each of the microservices, we used YAML definition files similar to the following example from the Python implementation of the inventory microservice:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: python-inventory
  labels:
    name: python-inventory
spec:
  replicas: 2
  selector:
    matchLabels:
      name: python-inventory
  template:
    metadata:
      name: python-inventory
      labels:
        name: python-inventory
    spec:
      containers:
```

```
- name: python-inventory
  image: iad.ocir.io/maacloud/maa-microservices/pythoninventory:v12[c1]
  ports:
    - containerPort: 8080
  resources:
    requests:
      memory: 256Mi
    limits:
      memory: 512Mi
  env:
    - name: DB_CONNECT_STRING
      value: "<DB_CONNECT_STRING>"
    - name: DB_USER
      value: "python_inventory"
    - name: DB_PASSWORD
      value: "<DB_PASSWORD>"
    - name: DB_CONNECTION_COUNT
      value: "5"
    - name: WORKERS
      value: "1"
    - name: HTTP_THREADS
      value: "16"
    - name: PORT
      value: "8080"
    - name: DEBUG_MODE
      value: "1"
    - name: AQ_CONSUMER_THREADS
      value: "1"
    - name: QUEUE_OWNER
      value: "PYTHON_AQ"
  readinessProbe:
    exec:
      command:
      - cat
      - /tmp/ready
    initialDelaySeconds: 0
    periodSeconds: 1
    timeoutSeconds: 1
    successThreshold: 1
```

```
            failureThreshold: 1
      imagePullSecrets:
      - name: <SECRET>
```

Microservice deployment on Kubernetes used the following command:

```
kubectl create -f app.yaml
```

The following is an example from the Python inventory microservice of the YAML definition for the load balancer:

```
apiVersion: v1
kind: Service
metadata:
  name: python-inventory-svc
spec:
  type: LoadBalancer
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    name: python-inventory
```

Load balancer deployment on Kubernetes used the following command:

```
kubectl create -f lb.yaml
```

Test Workload

Artillery enabled us to drive a test workload. We monitored and collected the output from the test runs.

TEST DATA GENERATION

The following Python program generated 10 separate CSV test data files which drove workload for the tests:

```
import csv
import random

itemids=["30","31","32","33","34"]
num_files = 10
for f in range(1, num_files+1):
    with open('order%d.csv' % f, 'w') as myfile:
        wr = csv.writer(myfile, quoting=csv.QUOTE_ALL)
        for i in range( f, 200001, num_files):
            wr.writerow(["%06d" % (i), random.choice(itemids)])
```

The following bash script shuffled (randomized) the data:

```
for ((i = 1 ; i <= 10 ; i++));
do
  export CSV_FILE=order${i}.csv
  shuf $CSV_FILE > shuf_$CSV_FILE
done
```

WORKLOAD

The Artillery YAML definitions described in the below table drove the three HTTP REST interfaces at a rate of 60 requests per second:

OPERATION	ARTILLERY YAML DEFINITION

PUT /placeOrder	`config:`
	`  environments:`
	`    python_kube:`
	`      target: 'http://150.136.194.42:8080'`
	`    python_docker:`
	`      target: 'http://0.0.0.0:8080'`
	`  payload:`
	`    path: "{{ $processEnvironment.CSV_FILE }}"`
	`    order: "sequence"`
	`    cast: false`
	`    fields:`
	`      - "orderid"`
	`      - "itemid"`
	`  phases:`
	`    - duration: 140`
	`      arrivalRate:  30`
	`      name: "PUT"`
	`scenarios:`
	`  - name: "PUT"`
	`    weight: 12`
	`    flow:`
	`      - put:`
	`          url: "/placeOrder?orderid={{ orderid }}&itemid={{ itemid }}`
	`&deliverylocation=London"`

GET /showOrder	`config:`
	`  environments:`
	`    python_kube:`
	`      target: 'http://150.136.194.42:8080'`
	`    python_docker:`

```
      target: 'http://0.0.0.0:8080'
  payload:
    path: "{{ $processEnvironment.CSV_FILE }}"
    order: "sequence"
    cast: false
    fields:
      - "orderid"
      - "itemid"
  phases:
    - duration: 120
      arrivalRate:  30
      name: "GET"
scenarios:
  - name: "GET"
    weight: 10
    flow:
      - get:
          url: "/showOrder?orderid={{ orderid }}"
```

GET /inventory

```
config:
  environments:
    python_kube:
      target: 'http://150.136.0.242:8080'
    python_docker:
      target: 'http://0.0.0.0:8081'
  payload:
    path: "inventory.csv"
    fields:
      - "inventoryid"
  phases:
    - duration: 120
      arrivalRate: 30
      name: "GET"
scenarios:
  - name: "GET"
    weight: 10
    flow:
      - get:
          url: "/inventory/{{ inventoryid }}"
```

The following bash script drove the workload:

```
ENV=$1

WORKERS=$2

FOLDER=$3

TEST=$4

PREFIX="$5"

set -e


mkdir -p $FOLDER

mkdir $FOLDER/$TEST


#PUT FIRST

for ((i = 1 ; i <= $WORKERS ; i++));

do

  sleep 0.01

  export CSV_FILE=${PREFIX}order${i}.csv

  artillery run -e $ENV put_art.yaml | tee -a $FOLDER/$TEST/put_art.txt &

done


sleep 20


#GET ORDER NEXT

for ((i = 1 ; i <= $WORKERS ; i++));

do

  sleep 0.01

  export CSV_FILE=${PREFIX}order${i}.csv

  artillery run -e $ENV get_ord_art.yaml | tee -a $FOLDER/$TEST/get_ord_art.txt &

done


#GET INVENTORY NEXT

for ((i = 1 ; i <= $WORKERS ; i++));

do

  sleep 0.01

  artillery run -e $ENV get_inv_art.yaml | tee -a $FOLDER/$TEST/get_inv_art.txt &

done


wait
```

JAVA CASE STUDY

Implementation

SOFTWARE VERSIONS

SOFTWARE	VERSION
Linux	oracle-epel-release-el7.x86_64 0:1.0-2.el7
Apache Maven	3.6.1
JDK	11.0
Helidon SE	1.4.3
Oracle Database and Grid Infrastructure	19.6
Kubernetes	1.14.8
Docker	18.09.8

IMPORTED LIBRARIES

Oracle JDBC drivers, ojdbc10.jar with companion jars, located in Maven Central:

- ojdbc10.jar: 19.3 JDBC Thin driver;
- ucp.jar: required for UCP(Universal Connection Pool);
- ons.jar: for use by the Oracle Notification Services (ONS) daemon;
- simplefan.jar: Java APIs for subscribing to RAC Fast Application Notification (FAN) events via ONS;
- xdb.jar: to support standard JDBC 4.x java.sql.SQLXML interface

Helidon's project pom.xml *pulls ojdbc10.jar* and its dependencies:

```
  <dependency>
      <groupId>com.oracle.ojdbc</groupId>
      <artifactId>ojdbc10</artifactId>
      <version>19.3.0.0</version>
    </dependency>
```

SODA for JAVA:

- https://github.com/oracle/soda-for-java/releases/download/v${SODA_VERSION}/orajsoda-${SODA_VERSION}.jar

Package lombok.extern.slf4j:

Lombok provides several log annotations to work with different logging libraries. In the end, they all generate a logger instance named as log. Slf4j generates a logger using the SLF4J API. c Slf4j library:

```
<!-- logging -->
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>${logback.version}</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>jul-to-slf4j</artifactId>
            <version>${jul-to-slf4j.version}</version>
        </dependency>
<!-- /logging -->
```

DATABASE CONNECTION POOLING

The microservice connected to the Oracle database using the Java Connection Pool.

UCP connection pool creation incorporated the following attributes specified in the Kubernetes YAML file:

- Initial, minimum and maximum connection counts controlled the size of the pool
- The FCF pool property enabled and disabled. FCF : dataSource.setFastConnectionFailoverEnabled(true);
- The following connection timeout parameters optimized UCP and addressed temporary connection shortage during outages: TimeoutChekInterval, InactiveConnectionTimeout, QueryTimeout and WaitTimeout

UCP data source factory declaration

```
dataSource = PoolDataSourceFactory.getPoolDataSource();
dataSource.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
dataSource=setUser(dbConfig.get("user").asString().get());
dataSource.setPassword(dbConfig.get("password").asString().get());
dataSource.setURL(dbConfig.get("url").asString().get());
```

FAN Enablement

```
Boolean boolParam = dbConfig.get("failoverEnabled").asBoolean().orElse(null);
dataSource.setFastConnectionFailoverEnabled(boolParam);
```

UCP Size

```
dataSource.setInitialPoolSize(intParam);
dataSource.setMinPoolSize(intParam);
dataSource.setMaxPoolSize(intParam);
```

UCP Optimization

```
dataSource.setTimeoutCheckInterval(intParam);
dataSource.setInactiveConnectionTimeout(intParam);
dataSource.setQueryTimeout(intParam);
dataSource.setConnectionWaitTimeout(intParam);
```

FUNCTIONAL IMPLEMENTATION

MICROSERVICE	OPERATION	JAVA CODE

Orders	PUT /placeOrder	`// DB  Transaction`
		`try {`
		`            database.getContext().execute(conn -> {`
		`                Order order = dao.create(conn, new Order(orderId, itemId, deliveryLocation));`
		`                String messageId = dao.pushToQueue(conn, order);`
		`                conn.commit();`
		`                conn.close();`
		`                }, log);`
		`        } catch (Exception e) {`
		`            errorOrdersCreate.inc();`
		`            request.next(e);`
		`        }`
		
		`// dao (Data Access Object)`
		`        OracleDatabase soda = database.getSoda().getDatabase(conn);`
		`        OracleCollection col = soda.openCollection(collectionName);`
		`        OracleDocument doc = soda.createDocumentFromString(order.getOrderid(),JsonUtils.writeValueAsString(order));`
		`        col.insert(doc);`
		`        log.debug("Created {}", order);`
		`        return order;`
		`public String pushToQueue(OracleConnection conn, Order order) throws SQLException {`
		`        return QueueUtils.sendMessage(conn, new OrderMessage(order), queue, QUEUE_SENDER_NAME, QUEUE_SENDER_ADDRESS);}`

	GET /showOrder	`// DB Transaction`
		`try {`
		`            database.getContext().execute(conn -> {`
		`                Order retval = dao.get(conn, request.queryParams().first("orderid").orElse(null));`
		`                conn.close();`
		`                if (Objects.nonNull(retval)) {`
		`                    response.send(JsonUtils.writeValueAsString(retval));`
		`                } else {`

		`throw new HttpStatusException(404, "Order not found.");`
		`}`
		`}, log);`
		`} catch (Exception e) {`
		`errorOrdersGet.inc();`
		`request.next(e);`
		`}`
		
		`// dao (Data Access Object)`
		`OracleDatabase soda = database.getSoda().getDatabase(conn);`
		`OracleCollection col = soda.openCollection(collectionName);`
		`OracleDocument doc = col.find().key(id).getOne();`
		`if (Objects.nonNull(doc)) {`
		`return JsonUtils.read(doc.getContentAsString(), Order.class);`
		`} else {`
		`return null;`
		`}`

	Inventory Queue Consumer	`this.consumer = new QueueConsumer(database, queue, MetricUtils.invQConsumer, (conn, message) -> {`
		`InventoryMessage inventory = JsonUtils.read(message, InventoryMessage.class);`
		`Order update = new Order(inventory.getOrderId(), null, null);`
		`if (OrderConsumer.EXISTS.equals(inventory.getState())) {`
		`update.setStatus("successful");`
		`update.setSuggestivesaleitem("suggestiveSaleItem");`
		`update.setDeliverylocation(inventory.getInventorylocation());`
		`} else {`
		`update.setStatus("failed to inventory");`
		`}`
		`dao.update(conn, update);`
		`conn.commit();`
		`}, log);`

Inventory	GET /inventory	`// DB Transaction`
		`try {`
		`database.getContext().execute(conn -> {`
		`Inventory retval = dao.get(conn, request.path().param("id"));`
		`conn.close();`
		`if (Objects.nonNull(retval)) {`

```
                    response.send(JsonUtils.writeValueAsString(retval));
                } else {
                    throw new HttpStatusException(404, "Inventory not 
found.");
                }
}, log);
        } catch (Exception e) {
            errorInventoryGet.inc();
            request.next(e);
}


// dao (Data Access Object)
try (OraclePreparedStatement st = (OraclePreparedStatement) conn.
prepareStatement(GET_BY_ID)) {
            st.setString(1, id);
            ResultSet res = st.executeQuery();
            if (res.next()) {
                return new Inventory(res);
            } else {
                return null;
}
}
```

Order Queue Consumer

```
// Order Consumer
this.consumer = new QueueConsumer(database, queue, MetricUtils.
ordQConsumer, (conn, message) -> {
            OrderMessage order = JsonUtils.read(message, OrderMessage.
class);
            String location = dao.decrement(conn, order.getItemId());
            String state;
            if (Objects.nonNull(location)) {
                state = EXISTS;
            } else {
                state = "inventorydoesnotexist";
            }
            dao.pushToQueue(conn, order.getOrderId(), state, location);
            conn.commit();
        }, log);


// dao (Data Access Object)
```

		`String DECREMENT_BY_ID = "update inventory set inventorycount = inventorycount - 1 where inventoryid = ? and inventorycount > 0 returning inventorylocation into ?";`
		
		`try (OraclePreparedStatement st = (OraclePreparedStatement) conn.prepareStatement(DECREMENT_BY_ID)) {`
		`            st.setString(1, id);`
		`            st.registerReturnParameter(2, Types.VARCHAR);`
		`            int i = st.executeUpdate();`
		`            ResultSet res = st.getReturnResultSet();`
		`if (i > 0 && res.next()) {`
		`                String location = res.getString(1);`
		`                log.debug("Decremented inventory id {} location {}", id, location);`
		`return location;`

DOCKER PACKAGING

IMAGE	DOCKER BUILD

orders	`# run + build just orders`
	`docker-compose -f project.yml up -d --build orders`

inventory	`# run + build just inventory`
	`docker-compose -f project.yml up -d --build inventory`

We used Docker Compose to define and run a multi-container Docker application including orders and inventory. project.yml is a docker compose yaml file that defines the order and inventory services.

Dockerfile:

```
FROM openjdk:11-buster

RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app

ADD ./target /usr/src/app

ENV JAVA_OPTS="$JAVA_OPTS"

ENV APP_VERSION=0.0.1

EXPOSE 8080

CMD exec java -jar $JAVA_OPTS /usr/src/app/helidon-$APP_VERSION.jar $APP_MODE
```

project.yml:

```
version: "3.3"
services:
  orders:
    image: iad.ocir.io/maacloud/maa-microservices/helidon-orders:v1.0.1
    build: ../
    ports:
      - "81:81"
    environment:
      - APP_MODE=orders
      - DB_URL=$DB_URL
      - LOG_LEVEL=DEBUG
  inventory:
    image: iad.ocir.io/maacloud/maa-microservices/helidon-inventory:v1.0.1
    build: ../
    ports:
      - "82:82"
    environment:
      - APP_MODE=inventory
      - DB_URL=$DB_URL
      - LOG_LEVEL=DEBUG
networks:
  default:
    ipam:
      config:
        - subnet: 10.100.0.0/24
```

Test Results

The implementation deployed in the test environment and tested under load functioned according to the specification.

PYTHON CASE STUDY

Implementation

SOFTWARE VERSIONS

SOFTWARE	VERSION
Linux	oracle-epel-release-el7.x86_64 0:1.0-2.el7
Python	3.6.8
cx_Oracle	7.3.0
Oracle Database and Grid Infrastructure	19.6
Oracle Instant Client	19.3
Kubernetes	1.14.8
Docker	18.09.8

IMPORTED LIBRARIES

- flask implements the REST interface to the service
- simplejson provides JSON formatting

MICROSERVICE CONFIGURATION

The following example code extracts parameters from the environment during execution:

```
db_connect_string = env.get('DB_CONNECT_STRING')
```

DATABASE CONNECTION POOLING

Connection pool creation used the following attributes:

- Minimum and maximum connection counts set to the same value to prevent connection storms
- events=True enabled FAN
- During outages there can be a temporary shortage of connections in the pool. The getmode and waitTimeout settings ensure that threads wait for connections to be recovered instead of failing.

```
pool = cx_Oracle.SessionPool(

          db_user,

          db_password,

          db_connect_string,

          encoding="UTF-8",

          min=db_connection_count,

          max=db_connection_count,

          threaded=True,
          events=True,
          getmode=cx_Oracle.SPOOL_ATTRVAL_TIMEDWAIT,
          waitTimeout=10000)
```

FUNCTIONAL IMPLEMENTATION

MICROSERVICE	OPERATION	PYTHON CODE

Orders	PUT /placeOrder	`soda = conn.getSodaDatabase()`
		`orderCollection = soda.openCollection("orders")`
		`orderDoc = soda.createDocument(order, key=order['orderid'])`
		`orderCollection.insertOne(orderDoc)`
		
		`orderQueue = conn.queue(queue_owner + ".orderqueue")`
		`orderQueue.enqOne(conn.msgproperties(payload = simplejson.dumps(`
		`   {'orderid': order["orderid"], 'itemid': order["itemid"]}`
		`)))`
		
		`conn.commit()`

	GET /showOrder	`orderCollection = conn.getSodaDatabase().openCollection("orders")`
		`orderDoc = orderCollection.find().key(request.args.get('orderid')).getOne()`

	Inventory Queue Consumer	`inventoryResponse = simplejson.loads(inventoryQueue.deqOne().payload)`
		
		`order = orderCollection.find()`
		`   .key(inventoryResponse["orderid"]).getOne().getContent()`
		
		`if inventoryResponse['action'] == 'inventoryexists':`
		`   order['status'] = 'successful'`
		`   order['suggestiveSaleItem'] = 'suggestiveSaleItem'`
		`   order['inventoryLocation'] = inventoryResponse['inventorylocation']`
		`else:`
		`   order['status'] = 'failed no inventory'`
		
		`orderCollection.find().key(inventoryResponse["orderid"]).replaceOne(order)`
		
		`conn.commit()`

Inventory	GET /inventory	`cursor.execute("select * from inventory where inventoryid = :id",`
		`   [inventory_id])`

	Order Queue Consumer	`sql = """update inventory set inventorycount = inventorycount - 1`
		`         where inventoryid = :inventoryid and inventorycount > 0`
		`         returning inventorylocation into :inventorylocation"""`
		
		`orderInfo = simplejson.loads(orderQueue.deqOne().payload)`
		
		`ilvar = cursor.var(str)`
		`ilvar.setvalue(0,"")`
		`cursor.execute(sql, [orderInfo["itemid"], ilvar])`
		`inventorylocation = ilvar.getvalue(0)`
		
		`inventoryQueue.enqOne(conn.msgproperties(`
		`   payload = simplejson.dumps(`
		`      {'orderid': orderInfo["orderid"],`
		`      'action': "inventoryexists" if cursor.rowcount == 1`
		`         else "inventorydoesnotexist",`
		`      'inventorylocation': inventorylocation}`
		`   )))`
		
		`conn.commit()`

DOCKER PACKAGING

IMAGE	DOCKER FILE

Oracle19.3_python	`FROM oraclelinux:7-slim`
	`ARG release=19`
	`ARG update=3`
	`RUN  yum -y install oracle-release-el7 && \`
	`     yum-config-manager --enable ol7_oracle_instantclient && \`
	`     yum -y install oracle-instantclient${release}.${update}-basiclite && \`
	`     yum install -y oracle-epel-release-el7 && \`
	`     yum install -y python36 && \`
	`     rm -rf /var/cache/yum`

orders
inventory
	`FROM oracle19.3_python`
	`WORKDIR /app`
	`ADD . /app`
	`RUN python3.6 -m pip install -r /app/requirements.txt`
	`CMD ["gunicorn", "app:app", "--config=config.py"]`

The requirements.txt file included the following libraries:

```
Flask_restful
gunicorn
cx_Oracle
simplejson
```

The following config.py code extracted parameters from the environment:

```
from os import environ as env

# Gunicorn Configuration
bind = ":" + env.get("PORT", "8080")
workers = int(env.get("WORKERS", 1))
threads = int(env.get("HTTP_THREADS", 1))
```

Test Results

The implementation deployed in the test environment and tested under load functioned according to the specification.

NODE.JS CASE STUDY

Implementation

SOFTWARE VERSIONS

SOFTWARE	VERSION
Linux	oracle-epel-release-el7.x86_64 0:1.0-2.el7
Node.js	10.19.0
node-oracledb	4.2.0
nvm	0.35.2
npm	6.13.7
express	4.17.1
express-validator	6.4.0
morgan	1.9.1
Oracle Database and Grid Infrastructure	19.6
Oracle Instant Client	19.5
Kubernetes	1.14.8
Docker	18.09.8

NODE LIBRARIES

- express provides a web server framework and enables JSON extraction from the request body
- express-validator provides input validation
- morgan provides logging
- node-oracledb provides the interface to the Oracle database

MICROSERVICE CONFIGURATION

We implemented most parameterization through environment variables, for example the following:

```
const webConfig = {
  port: process.env.HTTP_PORT || 8080
}
```

DATABASE CONNECTION POOLING

We created a database connection pool with the following attributes:

- Minimum (poolMin) and maximum (poolMax) connection counts set to the same value, per Oracle Real World Performance Team guidance to prevent connection storms, using the environment variable DB_CONNECTION_COUNT.
- Pool increment (poolIncrement) set to 0.
- Set events=true to enable FAN. This can be set via the oracledb.events property or at connection pool creation by setting the events property in the object passed for poolAttrs. By default events it is set to false at the oracledb level except for two specific versions (4.0.0 and 4.0.1) in which it was set to true by default. For the purposes of this testing, enabled setting the value via environment variable DB_M_FAN_EVENTS for the oracledb level.
- During outages there can be a temporary shortage of connections in the pool. Requests for connections are queued. The queueTimeout settings ensure that queued connection requests will wait this amount of time before failing. The default value is 60000 (60 seconds). For the purpose of this testing, enabled this to be set via the environment variable DB_CP_QUEUE_TIMEOUT, and tested in these tests with a value of 10000 (10 seconds).

```
const dbConfig = {
  orderPool: {
    user: process.env.DB_USER,
    password: process.env.DB_PASSWORD,
    connectString: process.env.DB_CONNECT_STRING,
    poolMin: Number(process.env.DB_CONNECTION_COUNT) || 10,
    poolMax: Number(process.env.DB_CONNECTION_COUNT) || 10,
    poolIncrement: process.env.DB_POOL_INC || 0
  }
};

switch (process.env.DB_M_FAN_EVENTS) {
  case 'false':
    console.log('Setting oracledb.events to false...');
    oracledb.events = false;
    break;
  case 'true':
    console.log('Setting oracledb.events to true...');
    oracledb.events = true;
    break;
  default:
    console.log('Keeping default value for oracledb.events...');
}

if (process.env.DB_CP_QUEUE_TIMEOUT) {
  console.log('Setting dbConfig.orderPool.queueTimeout to environment variable value [%s]...', process.env.
DB_CP_QUEUE_TIMEOUT);
  dbConfig.orderPool.queueTimeout = Number(process.env.DB_CP_QUEUE_TIMEOUT);
}
```

Because node-oracledb makes use of worker threads via the libuv library, it is important to ensure that there are at least as many worker threads as database connections. By default there are 4 worker threads. As worker threads could be required for other application processing if the application does both database and non-database work at the same time, for our testing we chose to increase the number of worker threads by the number of connections in the pool. We set this value by setting the environment variable UV_THREADPOOL_SIZE. For more information, see https://oracle.github.io/node-oracledb/doc/api.html#-143-connections-threads-and-parallelism.

```
const defaultUVThreadPoolSize = 4;

const currUVThreadPoolSize = process.env.UV_THREADPOOL_SIZE || defaultUVThreadPoolSize;

if (currUVThreadPoolSize < defaultUVThreadPoolSize + dbConfig.orderPool.poolMax) {

  process.env.UV_THREADPOOL_SIZE = dbConfig.orderPool.poolMax + defaultUVThreadPoolSize;

}
```

FUNCTIONAL IMPLEMENTATION

MICROSERVICE	OPERATION	NODE.JS CODE

Orders	PUT /placeOrder	`const soda = connection.getSodaDatabase();`
		`const sodaCollection = await soda.openCollection(sodaConfig.ordersCollectionName);`
		`const newOrderDoc = soda.createDocument(order, { key: order.orderid});`
		`createdDoc = await sodaCollection.insertOneAndGet(newOrderDoc);`
		
		`const orderQueue = await connection.getQueue(queueConfig.orderQueue);`
		`const orderMsgContent = {`
		`  orderid: order.orderid,`
		`  itemid: order.itemid`
		`}`
		`const inventoryMsg = await orderQueue.enqOne(JSON.stringify(orderMsgContent));`
		`await connection.commit();`

	GET /showOrder	`const soda = connection.getSodaDatabase();`
		`const sodaCollection = await soda.openCollection(sodaConfig.ordersCollectionName);`
		`const orderDoc = await sodaCollection.find().key(orderKey).getOne();`

	Inventory Queue Consumer	`const inventoryMsg = await inventoryQueue.deqOne();`
		
		`const inventoryMsgContent = JSON.parse(inventoryMsg.payload.toString());`

		`const soda = connection.getSodaDatabase();`
		`const sodaCollection = await soda.openCollection(sodaConfig.ordersCollectionName);`
		`const orderDoc = await sodaCollection.find().key(inventoryMsgContent.orderid).getOne();`
		`order = orderDoc.getContent();`
		`order.itemid = inventoryMsgContent.inventoryid;`
		`if (inventoryMsgContent.action === 'inventoryexists') {`
		`  order.status = 'successful';`
		`  order.suggestiveSaleItem = 'suggestiveSaleItem';`
		`} else {`
		`  order.status = 'failed no inventory';`
		`}`
		
		`if (inventoryMsgContent.location) {`
		`  order.inventorylocation = inventoryMsgContent.location;`
		`}`
		
		`const newOrderDoc = soda.createDocument(order, { key: inventoryMsgContent.orderid});`
		
		`const updatedOrderDoc = await sodaCollection.find().key(inventoryMsgContent.orderid).replaceOneAndGet(newOrderDoc);`
		
		`await connection.commit();`

Inventory	GET /inventory	`const sqlStatement =`
		`` `select inventoryid "inventoryid" ``
		`   , inventorylocation "inventorylocation"`
		`   , inventorycount "inventorycount"`
		`from inventory`
		`where 1=1`
		`` and inventoryid = :inventoryid`; ``
		
		`bindVariables.inventoryid = req.params.inventoryid;`
		
		`options.outFormat = oracledb.OUT_FORMAT_OBJECT;`
		
		`connection = await oracledb.getConnection();`
		
		`const queryResult = await connection.execute(sqlStatement, bindVariables, options);`

	Order Queue Consumer	

```
const orderMsg = await orderQueue.deqOne();

const orderMsgContent = JSON.parse(orderMsg.payload.
toString());


const updateSQL =

`update inventory

   set inventorycount = inventorycount - 1

 where 1=1

   and inventoryid = :inventoryid

   and inventorycount > 0

 returning inventorylocation

      into :inventorylocation`;


if ((orderMsgContent) && (orderMsgContent.itemid) &&
(orderMsgContent.orderid)) {

  bindVariables.inventoryid = orderMsgContent.itemid

}


bindVariables.inventorylocation = {

  dir: oracledb.BIND_OUT,

  type: oracledb.STRING

};


options.outFormat = oracledb.OUT_FORMAT_OBJECT;


const queryResult = await connection.execute(updateSQL,
bindVariables, options);


if (queryResult.rowsAffected && queryResult.rowsAffected
=== 1) {

  action = "inventoryexists";

  location = queryResult.outBinds.inventorylocation[0];

} else {

  action = "inventorydoesnotexist";

  location = "";

}


const inventoryMsgContent = {

  orderid: orderMsgContent.orderid,

  action: action,
```

		```
		  location: location
		};

		const inventoryMsg = await inventoryQueue.enqOne(JSON.
		stringify(inventoryMsgContent));

		await connection.commit();
		```

DOCKER PACKAGING

IMAGE	DOCKER FILE

orders
inventory
	```
	FROM oraclelinux:7-slim

	RUN  yum -y install oracle-release-el7 oracle-nodejs-release-el7 && \
	     yum-config-manager --disable ol7_developer_EPEL && \
	     yum -y install oracle-instantclient19.5-basiclite nodejs && \
	     rm -rf /var/cache/yum

	# Create app directory
	WORKDIR /usr/src/orders

	# Install app dependencies
	# A wildcard is used to ensure both package.json AND package-lock.json are copied
	# where available (npm@5+)
	COPY package*.json ./

	RUN npm install
	# If you are building your code for production
	# RUN npm ci --only=production

	# Bundle app source
	COPY . .

	EXPOSE 8080
	CMD [ "node", "app.js" ]
	```

Test Results

The implementation deployed in the test environment and tested under load functioned according to the specification.

CONCLUSION

Through the case studies, we have demonstrated how data centric microservices conforming to our simple architecture and best practices, built in Java Helidon, Python and Node.js, and deployed on Oracle Cloud Infrastructure and Oracle Database, functioned exactly to our specification. Further we showed the flexibility of the Oracle Converged Database to store different data types making the data architecture simple.