Microservices, Containers, Databases and Persistence Models

Kuassi Mensah
Director Product Management, Oracle

Paul Parkinson
Consultant Member Technical Staff, Oracle

October 23, 2018
Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing, and pricing of any features or functionality described for Oracle’s products may change and remains at the sole discretion of Oracle Corporation.
Agenda

• Concepts: Microservices, Containers, Kubernetes
• Transaction Models
• Microservices Persistence and Databases
• Demo
Concepts: Microservices, Containers, Kubernetes
MicroServices

Build from Scratch or Break Monolithic apps into Modules

• Reduce complexity
• Module specific data model
• Scale at module level
• Faster Development
  – API-First, Events-First, Domain Driven Design
• Blue/green deployments
• Idempotent
• Fault Isolation
Containers / Docker

- OS virtualization
- Modularity
- Reduced size
- Greater scaling
- Isolated environment for running apps, webserver, persistence store, etc
- Consistency in runtime environments (including testing and production)
- Operational simplicity
- Productivity
- Enable portability from on-premises to the Cloud

Source: Docker
Containers Orchestration / Kubernetes

Containers must be managed (resource), scheduled, load balanced and connected to external world

*Kubernetes*: Greek word for a ship's captain

• Goal: robust platform for running thousands of containers in production.
• System for automating deployment and scaling of containerized apps
• Velocity – constant updates, pause/resume deployments, version control
• Immutability – new image for changes (maintains history, lineage)
• Declarative API & config – describe desired state of system, abstracts “how”
• Self healing – continuously maintains desired state over time.
• Auto scaling
Kubernetes/MicroServices

Ideal for microservices...

- Pods, a runnable unit of work (container images developed by different teams into a single deployable unit); usually holds 1 or 2 containers
- Load balancing, naming and discovery isolate one microservice from another
- Namespaces provide isolation and access control so that each microservice can control the degree to which other services interact with it.
- Platform services tells the rest of the Kubernetes environment what services your application provides.
- Api-registry: tracks of all available services and the resources they expose.
Kubernetes Platform Services on OKE
## Pain Points Addressed

<table>
<thead>
<tr>
<th>Pain Point</th>
<th>Addressed by</th>
</tr>
</thead>
<tbody>
<tr>
<td>All inter-service communication is subject to traditional networking</td>
<td>istio – transparent resiliency features like automatic re-ties, bulkheads reduce the impact of network</td>
</tr>
<tr>
<td>issues like latency and connection failures</td>
<td>issues</td>
</tr>
<tr>
<td>Need to update all client services when a new server service version is</td>
<td>istio – route rules allow traffic flows to be defined outside the services</td>
</tr>
<tr>
<td>deployed</td>
<td></td>
</tr>
<tr>
<td>Need service to service authorization and simple policy management for</td>
<td>speedle policy – supports auto discovery of authorization policies and simple policy language</td>
</tr>
<tr>
<td>zero trust networks</td>
<td></td>
</tr>
<tr>
<td>Large workflow made of many services may become brittle, resistant to</td>
<td>event broker – connecting services through events allows workflows to be defined outside of individual</td>
</tr>
<tr>
<td>adaptations</td>
<td>services</td>
</tr>
<tr>
<td>Widely distributed, multi-process workflows are difficult to diagnose and</td>
<td>sauron – single view for health of entire system</td>
</tr>
<tr>
<td>monitor</td>
<td></td>
</tr>
<tr>
<td>Deploying a service to different environments requires rebuilding the</td>
<td>atp dbaaS – provisioning through a service broker means the same service can pick up appropriate</td>
</tr>
<tr>
<td>service to use appropriate database connection</td>
<td>database connection information from cluster secret without</td>
</tr>
</tbody>
</table>
Istio Routing Rules

- Istio Route Rules can be used to change which service receives requests.
- This allows invoker to pass requests to different service with no code change or redeployment.
- Route rules can shift traffic all at once, or route percentages to new services.

RouteRule.yaml

```yaml
kind: VirtualService
metadata:
  name: process
namespace: demo
spec:
  hosts:
  - process
  http:
    - route:
      - destination:
        host: process
        subset: v2
```

```yaml
kind: DestinationRule
metadata:
  name: process
namespace: demo
spec:
  host: process
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```
Event Broker

- Adds rules, callbacks, and structure to vanilla message bus
- Rules are simple and straightforward
- Rules can be updated to alter request flow
- EB can leverage any message bus implementation underneath

EventRule.yaml

apiVersion: auraevents.oracledx.com/v1alpha1
kind: Rule
metadata:
  name: rule-process
  labels:
    auraevents.oracledx.com/eventType: process
spec:
  action:
    url: http://process.demo:2222
    callbackURL: http://storefront.sf:8888/callback
Event Broker

- CloudEvents standard

Example CloudEvent

```json
{
  "cloudEventsVersion": "0.1",
  "eventType": "com.example.someevent",
  "source": "/mycontext",
  "eventId": "A234-1234-1234",
  "eventTime": "2018-04-05T17:31:00Z",
  "comExampleExtension1": "value",
  "comExampleExtension2": {
    "otherValue": 5
  },
  "contentType": "text/xml",
  "data": "<much wow="xml"/>
}
```
Sauron Integration

• Easy integration of Sauron Managed Service
• Provides visual representation of application and system metrics
• Provides alerting features via eMail, Slack, etc.

Prometheus config

- job_name: 'events-broker'
  scrape_interval: 5s
  scrape_timeout: 5s
  static_configs:
  - targets: ['129.213.46.56:30350']

• Business metrics based on smart filtering of event metrics in dashboard
Marketplace Provisioning of ATP DBaaS

• Service Broker adds connection info into cluster via Kubernetes Secrets
• Service is written to consume Secret
• This allows service to be deployed in different environments without any code changes, and pick up the appropriate DB connection automatically

```yaml
Deployment.yaml
env:
- name: DB_ADMIN_USER
  valueFrom:
    secretKeyRef:
      name: atp-demo-binding
      key: user_name
- name: DB_ADMIN_PWD
  valueFrom:
    secretKeyRef:
      name: atp-user-cred
      key: password
- name: WALLET_PWD
  valueFrom:
    secretKeyRef:
      name: atp-user-cred
      key: wallet_password
volumeMounts:
- name: creds
  mountPath: /db-demo/creds
```
Transactions Models
MicroServices Integration

• An application become an assembly of MicroServices that communicate among themselves

• MS integration choices: file system, database, RPC, Event

• Event-based Integration: best choice
  – Asynchronous, loose coupling
  – Pub/Sub, Notification
  – State transfer
  – Immutable
  – Support for Event Sourcing and Command Query Responsibility Segratation (CQRS)
Event Sourcing

Microservices interact by ”sourcing events” in/out from the Event store via the Event Broker

• Event store (Kafka, AQ in the future or vanilla Oracle DB) is the single source of truth:
• Producers: log events in the Event store then publish a notification
• Consumers: notified when Events are published then “read” the Event store
Command Query Responsibility Segregation - CQRS

Separation of responsibilities within a bounded context (Design domain)

- Commands: make changes to state/data -- do not query state
- Queries: view state/data (scale with Materialized Views) -- do not change state
- Often combined with Event Sourcing
Business Transactions – Requirements

Distributed Transactions that span multiple microservices
   – E.g., Booking a trip include: flight, hotel, car, shows

• Microservice A calls B that calls C.
   – The workflow or business transaction commits only when C completes

• How do you undo A & B should C fail?
   – Traditional two-phase COMMIT TX is an anti-pattern with MicroServices
   – Each service commits individually (locally): how to ensure consistency?
     • SAGA Pattern
     • High concurrency with Escrow/Promises
Business Transactions Consistency with SAGA Pattern

“A saga is a sequence of local transactions. Each local transaction updates the state (database) and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.” https://microservices.io/patterns/data/saga.html

Two approaches for compensating failed member(s) of a distributed Tx

• Choreography
  – The services interact with each other to coordinate well known/defined activity (the demo has an example)

• Orchestration
  – An orchestrator is used to coordinate the activity.
  – All messages are passed through (intercepted and issued by) this orchestrator.

• Compensation can simply be Tx ABORT (this can be automated)
Business Transactions Consistency with Escrow/Promises

Escrow – long running business transactions

- Counters (any column with “commutative” properties) used for inventories management
  - Stock on hand, money/share/credit of account, available seats, ...
  - manipulated with plus/minus -- commutative
  - One request is sufficient \( \text{amount} := \text{amount} +/\!- \text{change} \)

- Queries return two distinct states: # available in stock, # in flow

- Benefits: parallel update, automatic compensation (the inverse function)

Promises (not Java/NodeJS promises)

- A promise is a granted escrow request with time limits (i.e., On-hold reservation)
  - Expiration needs careful handling of race conditions
Microservices Persistence and Databases

Data Models, Domains, Databases
Polyglot or Multi-Model Database?

Microservices architecture goals: flexibility and agility

• Polyglot system
  – a specialized database for each microservice
  – may lead to an expensive and difficult platform to operate

• Multi-model database (i.e., Autonomous Transaction Processing)
  – Each service gets or share an ATP service instance
  – high availability and durability options
Oracle Autonomous Database

Self-Driving
• Automates all monitoring, management and tuning
• Creates mission critical scale-out database with DR
• Full compatibility enables simple database migration

Self-Securing
• Automatically applies security updates online
• Secure configuration with full database encryption
• Sensitive data hidden from Oracle or customer admins

Self-Repairing
• Recovers automatically from any failure
• 99.995% uptime including maintenance, guaranteed
• Elastically scales compute or storage as needed
PDB Sharding for Microservices

Scalability, fault isolation and geo-distribution

- PDB Sharding in DB 19c
  - Each PDB can be sharded across multiple CDBs
- Provides fault isolation and geo-distribution for microservices
  - Loss of an entire CDB makes only part of a PDB unavailable
- Also allows each microservices to scale its PDB individually
  - More efficient use of resources compared to scaling a monolithic application (CDB).
Online Shopping
Kafka as Event Broker and Store
AQ Support for Microservices

- AQ/ATP as Event store
- Queue operations and DML Data and message in the same local transaction
- Sharded Queues furnish high scalability
- Transacted session guarantees no loss of message and exactly once delivery
- Client initiated notification
- Handles large message backlogs
- Queue level access privileges (enqueue, dequeue) supports CQRS
- AQ support for SAGA-pattern distributed transactions
- High Availability and Disaster Recovery
Online Shopping
Kafka Event Broker & Store + AQ Message provider service

Catalog ➔ Cart ➔ Order ➔ Inventory ➔ Shipping

Browse Catalog ➔ Product In Cart ➔ AQ Event Provider Service ➔ Place Order (insert db + AQ message) ➔ Cloud Event

K8 Event Broker and Event Rules

Kafka / Event Store / Event Log
AQ Enhancements to Support MicroServices

• Retention support in Sharded Queues
• Cloud Events support
• Log compaction
• Autonomous Queues in DBaaS
• Change Query Notification support in ATP using AQ secure notification
• Lightweight, seekable and retroactive subscribers
• Topic auto-creation
• Kafka API (Java)
Online Shopping
AQ as Event Broker and Store

Catalog

Cart

Order

Inventory

Shipping

Browse Catalog
Product In Cart

Place Order
(insert db + AQ message)

AQ Event Provider Service

AQ Broker, Store and Event Rules

Kafka / Event Store / Event Log

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.
Asynchronous Database Access for MicroServices

Microservices architecture goals: flexibility and agility

• Synchronous DB Access (JDBC)
  – The microservice waits till data update (DML) is completed
  – Publishes the event

• Asynchronous Db Access (ADBA)
  – The microservice submits a DML operation + AQ event publishing then moves on.
    a) Create a transaction object
    b) create/submit a RowCountOperation to insert an order
    c) get a CompletionStage
    d) create another RowCountOperation to publish an AQ event
    e) session.commitMaybeRollback(trans)
    f) submit the whole transaction and return

  – Hands-on lab and Session @ https://bit.ly/2DzV3uT
Refactoring Monolithic

• Captures a SQL Tuning Set (STS) while a SQL workload is running
• Produces a Map of SQL vs Tables
• Generates a Graph
Refactoring Monolithic – Bounded Contexts

• Identify non-conflicting data model
• Define services within Bounded Contexts boundaries
Online Shopping

Catalog

Cart

Order

Inventory

Supplier

Browse Catalog

Product In Cart

Place Order (insert db + AQ message)

Cloud Event Provider Service

K8 Event Broker and Event Rules

Kafka / Event Store / Event Log
Online Shopping - Workflow

1. Catalog service: Graph data model – not implemented
2. Order service: enters an order in the DB and publishes an AQ message in the same local Tx
3. Db event provider: picks up the AQ message publishes it to the Cloud Event Broker
4. Supplier service: supplies the Inventory service and publishes an AQ message in the same local Tx.
5. Inventory service: receives an Order place event then publishes a “lack of inventory for order item” event
6. Order service: picks up that event and constructs its state from the event sourcing mechanism w/o querying the database
- name: creds
  mountPath: /creds

containers:
- name: atp-inventory
  image: docker.io/paulparkinson/atp-inventory:latest
  imagePullPolicy: Always
  resources:
    requests:
      cpu: 100m
      memory: 100Mi
  env:
    - name: DB_ADMIN_USER
      valueFrom:
        secretKeyRef:
          name: atp-demo-binding
          key: user_name
    - name: DB_ADMIN_PWD
      valueFrom:
        secretKeyRef:
          name: atp-user-cred
          key: password
    - name: WALLET_PWD
      valueFrom:
        secretKeyRef:
          name: atp-user-cred
          key: wallet_password
  volumeMounts:
  - name: creds
    mountPath: /db-demo/creds
  ports:
  - containerPort: 8080
volumes:
- name: creds
  secret:
    secretName: atp-demo-binding
- name: creds
  emptyDir: {"}

public static OracleDataSource getDataSource() throws SQLException {
    OracleDataSource ds = new OracleDataSource();
    ds.setURL("jdbc:oracle:thin:@" + System.getProperty("tns_name"));
    ds.setUser(System.getenv("DB_ADMIN_USER"));
    ds.setPassword(System.getenv("DB_ADMIN_PWD"));
    return ds;
}
<table>
<thead>
<tr>
<th>NAMESPACE</th>
<th>NAME</th>
<th>DESIRED</th>
<th>CURRENT</th>
<th>UP-TO-DATE</th>
<th>AV</th>
</tr>
</thead>
<tbody>
<tr>
<td>aura-eb</td>
<td>events-broker-apiserver</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>aura-eb</td>
<td>events-broker-callback</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>aura-eb</td>
<td>events-broker-dispatcher</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>aura-eb</td>
<td>events-broker-invoker</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>catalog</td>
<td>atp-demo</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>catalog</td>
<td>catalog-catalog-apiserver</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>catalog</td>
<td>catalog-catalog-controller-manager</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>installer-operator</td>
<td>installer-operator</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>istio-system</td>
<td>grafana</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>istio-system</td>
<td>istio-citadel</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>istio-system</td>
<td>istio-egressgateway</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>istio-system</td>
<td>istio-galley</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>istio-system</td>
<td>istio-ingressgateway</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>istio-system</td>
<td>istio-pilot</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>istio-system</td>
<td>istio-policy</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>istio-system</td>
<td>istio-sidecar-injector</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>istio-system</td>
<td>istio-statsd-prom-bridge</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>istio-system</td>
<td>istio-telemetry</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>istio-system</td>
<td>istio-tracing</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>istio-system</td>
<td>kiali</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>istio-system</td>
<td>prometheus</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>kube-system</td>
<td>kube-dns</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>kube-system</td>
<td>kube-dns-autoscaler</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>kube-system</td>
<td>kubernetes-dashboard</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>kube-system</td>
<td>tiller-deploy</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>service-broker</td>
<td>oci-service-broker-oci-service-broker</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>txdemo</td>
<td>oracledb-eventprovider</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>txdemo</td>
<td>oracledb-inventoryscale</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>txdemo</td>
<td>oracledb-microservice-frontend</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>txdemo</td>
<td>oracledb-orderservice</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>txdemo</td>
<td>oracledb-queryml-catalog</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>txdemo</td>
<td>oracledb-soda-shoppingcart</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Oracle DB microservices demo

Catalog (demonstrates Graph QL against graph data model)

Search catalog for pencil(s)

Shopping Cart (demonstrates Simple Oracle Document Access against JSON model)

Get shopping cart

Add (pencil) to shopping cart

Empty shopping cart

Submit Order (demonstrates saga, event broker, and dupe-elim/idempotent nature of microservices)

order66

place order

Order Detail (demonstrates event sourcing and CQRS view (aggregate query) across item detail, order status, etc.)

show order

Backend/Admin (demonstrates saga, event broker)

Add Inventory

Remove Inventory

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.
Oracle DB microservices demo

Catalog (demonstrates Graph QL against graph data model)

Search catalog for pencil(s)

JSON...

```
[ {
  "item_id": "px11",
  "item_name": "pencil",
  "user_rating_range": 5,
  "seller": "schoolsuppliescompany"
}, {
  "item_id": "px9",
  "item_name": "pencil",
  "user_rating_range": 5,
  "seller": "pencompany"
}, {
  "item_id": "px12",
  "item_name": "pen",
  "user_rating_range": 5,
  "seller": "schoolsuppliescompany"
}]
```

table...

<table>
<thead>
<tr>
<th>item_id</th>
<th>item_name</th>
<th>user_rating</th>
<th>seller</th>
</tr>
</thead>
<tbody>
<tr>
<td>px11</td>
<td>pencil</td>
<td>5</td>
<td>schoolsuppliescompany</td>
</tr>
<tr>
<td>px9</td>
<td>pencil</td>
<td>4</td>
<td>pencompany</td>
</tr>
<tr>
<td>px12</td>
<td>pen</td>
<td>5</td>
<td>schoolsuppliescompany</td>
</tr>
</tbody>
</table>

Shopping Cart (demonstrates Simple Oracle Document Access against JSON model)

Get shopping cart

Add (pencil) to shopping cart

Empty shopping cart

Submit Order (demonstrates saga, event broker, and dupe-elim/idempotent)

Submit order
Oracle DB microservices demo

Catalog (demonstrates Graph QL against graph data model)

Search catalog for pencil(s)

Shopping Cart (demonstrates Simple Oracle Document Access against JSON model)

Get shopping cart

Add (pencil) to shopping cart

Empty shopping cart

Submit Order (demonstrates saga, event broker, and dupe-elim/idempotent nature of microservices)

order66

place order

Order Detail (demonstrates event sourcing and CQRS view (aggregate query) across item detail, order status, etc.)

show order

------------------

Backend/Admin (demonstrates saga, event broker)

Add Inventory

Remove Inventory
@RequestMapping("/adddtoshoppingcart")
public Map addDtoShoppingCart() {
    System.out.println("--- SODA ShoppingCart addDtoShoppingcart");
    try {
        OracleDatabase db = getOracleDatabase();
        OracleCollection col = db.openCollection("customerId1cartJSONCollection");
        if (col == null) col = db.admin().createCollection("customerId1cartJSONCollection");
        addDocItemsToCart(col, db);
        return getCartContentsForQuery(db, col);
    }
    catch (Exception e) {
        e.printStackTrace();
        return singletonMap("message", "shoppingcart exception:" + e);
    }
}

private void addDocItemsToCart(OracleCollection col, OracleDatabase db) throws SQLException {
    col.insert(db.createDocumentFromString("{ "name" : "item" + counterForItem++ + "foo", "price" : "50" }");
}

private String getJSONString() {
    return "{"name" : "item" + counterForItem++ + "foo", "price" : "50" }";
}

private Map getCartContentsForQuery(OracleDatabase db, OracleCollection col) throws SQLException {
    OracleDocument fetchedDoc;
    System.out.println("--- SODA ShoppingCart Retrieving documents...
    OracleCursor c = null;
    try {
        c = col.find(), getCursor();
        String contentAsString = ""
        while (c.hasNext()) {
            fetchedDoc = c.next();
            contentAsString = fetchedDoc.getContentAsString();
        }
    }
}
Oracle DB microservices demo

Catalog (demonstrates Graph QL against graph data model)

Search catalog for pencil(s)

Shopping Cart (demonstrates Simple Oracle Document Access against JSON model)

Get shopping cart

JSON...
{"shoppingcartcontents": {"name": "item4foo", "price": "50"}}

table...

<table>
<thead>
<tr>
<th>name</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>item4foo</td>
<td>50</td>
</tr>
</tbody>
</table>

Add (pencil) to shopping cart

Empty shopping cart

Submit Order (demonstrates saga, event broker, and dupe-elim/idempotent)
Oracle DB microservices demo

Catalog (demonstrates Graph QL against graph data model)

Search catalog for pencil(s)

Shopping Cart (demonstrates Simple Oracle Document Access against JSON model)

Get shopping cart

Add (pencil) to shopping cart

Empty shopping cart

Submit Order (demonstrates saga, event broker, and dupe-elim/idempotent nature of microservices)

order66

place order

Order Detail (demonstrates event sourcing and CQRS view (aggregate query) across item detail, order status, etc.)

show order

Backend/Admin (demonstrates saga, event broker)

Add Inventory

Remove Inventory
Report of failed order to client (the result of missing inventory)...

**Oracle DB microservices demo**

Catalog (demonstrates Graph QL against graph data model)

*Search catalog for pencil(s)*

Shopping Cart (demonstrates Simple Oracle Document Access against JSON model)

*Get shopping cart*

*Add (pencil) to shopping cart*

*Empty shopping cart*

Submit Order (demonstrates saga, event broker, and dupe-elim/idempotent nature of microservices)

```
order66
place order
```

JSON...

```
{"message":"placeorder insertOrderAndSendMessage complete for orderid:order66 orderStatus:failed"}
```

Table...

<table>
<thead>
<tr>
<th>orderid</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>order66</td>
<td>failed</td>
</tr>
</tbody>
</table>
Order inserted in table and AQ message sent in same local transaction (src code)...

```java
public static javax.jms.Queue insertOrderAndSendMessage(String message) {
    try {
        System.out.println("-----\tOracleDBController insertOrderAndSendMessage enter\t");
        OracleDataSource aqDataSource = getAQDataSource();
        QueueConnectionFactory q_cf = AQjmsFactory.getQueueConnectionFactory(aqDataSource);
        QueueConnection q_conn = q_cf.createQueueConnection();
        Session session = q_conn.createQueueSession(true, Session.CLIENT_ACKNOWLEDGE);
        Connection dbConnection = ((AQjmsSession) session).getDBConnection();
        System.out.println("-----\tOracleDBController insertOrderAndSendMessage dbConnection: " + dbConnection.createStatement().executeUpdate("insert into orders values (" + message + ", myinventoryid1, 1, 'mystatus1')
            Queue queue = ((AQjmsSession) session).getQueue(queueOwner, queueName);
            ((AQjmsDestination) queue).start(session, true, true);
            QueueSender sender = ((AQjmsSession) session).createSender(queue);
            Message msg = session.createTextMessage(message);
            sender.send(msg);
            session.commit();
            System.out.println("-----\tOracleDBController insertOrderAndSendMessage committed: " + message);
            session.close();
            q_conn.close();
            return queue;
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
```
Order inserted in table and AQ message sent in same local transaction (logs)...
AQ Event provider service

Mediating AQ message received from it into a Cloud Event that is published to event broker for processing based on rules.
There can be multiple rules for a single event type (e.g., order completion triggers email, delivery, etc. notification)

<table>
<thead>
<tr>
<th>frontend</th>
<th>catalog</th>
<th>shoppingcart</th>
<th>order</th>
<th>eventprovider</th>
<th>inventory</th>
</tr>
</thead>
</table>

```bash
pparkins-mac:oracledb-eventprovider pparkins$ kubectl logs oracledb-eventprovider-6696b66b56-rg8rb
-n txdemo

-----> OracleDBEventProvider starting...
-----> OracleDBEventProvider listenForMessages before receive
-----> OracleDBEventProvider listenForMessages after receive textMsg:order66
-----> OracleDBEventProvider publishing event to broker: order66
OracleDBEventProvider.publishEventToBroker urlString:http://129.146.94.229:80/events
OracleDBEventProvider.publishEventToBroker input:{"source":"oracledb","eventType":"orderplaced","eventID":"orderplacedid1","cloudEventsVersion":"0.1","data":{"orderid":"order66","inventoryid":"inventoryid1"}}
{"eventType":"orderplaced","cloudEventsVersion":"0.1","source":"oracledb","eventID":"orderplacedid1","data":{"inventoryid":"inventoryid1","orderid":"order66"}}
-----> OracleDBEventProvider published event to broker: order66
-----> OracleDBEventProvider listenForMessages before receive
```
# rules received by inventory service
apiVersion: auraevents.oracledx.com/v1alpha1
kind: Rule
metadata:
  name: orderplacedinventoryrule
  labels:
    auraevents.oracledx.com/eventType: orderplaced
spec:
  action:
    url: http://oracledb-inventorieservice.txdemo/orderplaced

# rules received by order service for event sourcing and cQRS
apiVersion: auraevents.oracledx.com/v1alpha1
kind: Rule
metadata:
  name: addinventoryorderrule
  labels:
    auraevents.oracledx.com/eventType: addinventory
spec:
  action:
    url: http://oracledb-orderservice.txdemo/adjustinventoryIncrease

apiVersion: auraevents.oracledx.com/v1alpha1
kind: Rule
metadata:
  name: removeinventoryorderrule
  labels:
    auraevents.oracledx.com/eventType: removeinventory
spec:
  action:
    url: http://oracledb-orderservice.txdemo/adjustinventoryDecrease

# rule received by order service for success
apiVersion: auraevents.oracledx.com/v1alpha1
kind: Rule
metadata:
  name: inventoryexistsorderrule
Inventory service receiving event as a result of rule processing of order placed Cloud event and then publishing event indicating (lack of) inventory for order item.
**Order service**

**receiving inventory event** and cancelling order as a result (setting status to failed and returning it to the client).

Choreography saga: services coordinate directly with each other to maintain data integrity for a business activity.
E.g., the lack of inventory (in the inventory service/database) in this case results in a cancelled/failed order (in the order service/database).
Supplier service
interface to inventory service adds inventory...
Report of failed order to client (the result of inventory now existing)...

Oracle DB microservices demo

Catalog (demonstrates Graph QL against graph data model)

Search catalog for pencil(s)

Shopping Cart (demonstrates Simple Oracle Document Access against JSON model)

Get shopping cart

Add (pencil) to shopping cart

Empty shopping cart

Submit Order (demonstrates saga, event broker, and dupe-elim/idempotent nature of microservices)

Submit Order

order67

place order

JSON...

{"message":"placeOrder insertOrderAndSendMessage complete for orderid:order67 orderStatus:successful"}

table...

<table>
<thead>
<tr>
<th>orderid</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>order67</td>
<td>successful</td>
</tr>
</tbody>
</table>
Same message/workflow and order is now successful as inventory exists.
Oracle DB microservices demo

Catalog (demonstrates Graph QL against graph data model)

Search catalog for pencil(s)

Shopping Cart (demonstrates Simple Oracle Document Access against JSON model)

Get shopping cart

Add (pencil) to shopping cart

Empty shopping cart

Submit Order (demonstrates saga, event broker, and dupe-elim/idempotent nature of microservices)

order66

place order

Order Detail (demonstrates event sourcing and CQRS view (aggregate query) across item detail, order status, etc.)

show order

Backend/Admin (demonstrates saga, event broker)

Add Inventory

Remove Inventory
CQRS (aggregate)
query providing data gleaned from event sourcing.
Data is sourced from events received and each service builds and accesses the specific state/representation it requires locally allowing for Command Query Responsibility Segregation.
The order service order detail, eg, in effect has information from other services’ databases (inventory and suggested items).

Order Detail (demonstrates event sourcing and CQRS view (aggregate query) across item detail, order status, etc.)

show order

JSON...
{"orderid": "order67", "orderstatus": "successful", "shippingid": "shippingid134", "suggesteditem": "px9", "inventorylocation": "austin" }

table...

<table>
<thead>
<tr>
<th>orderid</th>
<th>orderstatus</th>
<th>shippingid</th>
<th>suggesteditem</th>
<th>inventorylocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>order67</td>
<td>successful</td>
<td>shippingid134</td>
<td>px9</td>
<td>austin</td>
</tr>
</tbody>
</table>


Q and A
See our Next Sessions @ https://bit.ly/2DzV3uT