Microservices Workshop Lab Guide

Chris Richardson

Table of Contents

In the workshop you will run a really simple Order Management application that demonstrates the key ideas in the Microservice architecture. The application consists of three services. The services communicate using Redis Streams.

Key patterns include:

- Microservice architecture - decompose a system into loosely coupled services
- Database per service - essential for ensuring loose coupling
- Saga - implement transactions that span multiple services
- CQRS - implement queries that retrieve data from multiple services
- Distributed tracing - tracks and visualizes how requests flow between the services

1. Lab setup

PLEASE NOTE It’s important that you complete this setup guide before the workshop. It requires significant time and network bandwidth. If you get stuck please post a question to the Slack workspace.

The key steps:

1. Install Docker
2. Get the source code for the labs
3. Compile and start the microservices to ensure that dependencies are downloaded and verify that everything works

1.1. Requirements

These labs require Internet access to download Java libraries and Docker images.

A laptop/desktop (ideally with at least 16G of memory) with the following installed:

- Docker for Windows/Mac.
- Git (or you can download the ZIP from github)
- An editor. Preferably your favorite Java development IDE, e.g. Intellij IDEA or Eclipse, with the Gradle plugin installed. However, a text editor is sufficient.

1.1.1. Installing Docker

The version of Docker to install depends on whether you are using Windows, or Mac OS. It also depends on which version of Windows you are running.

- Windows:
  - Please read this first for a discussion of Windows versions
  - Modern versions of Windows - Docker for Windows
  - Older versions of Windows - Docker Toolbox
- Mac OS - Docker for Mac

If you are running in a corporation that requires using an HTTP(S) proxy please set the appropriate environment variables (e.g. HTTP_PROXY, HTTPS_PROXY, NO_PROXY) to configure proxy usage

1.2. Getting the source code for the labs

Git clone the eventuate-tram-examples-customers-and-orders-redis repository:

```
git clone git@github.com:eventuate-tram/eventuate-tram-examples-customers-and-orders-redis.git
```

or

```
git clone https://github.com/eventuate-tram/eventuate-tram-examples-customers-and-orders-redis.git
```

Note: you can also download the ZIP from github.

1.3. Running the commands

There are a couple of different ways to run the various commands to compile the Java services and run them.

1.3.1. Easier: Container-based command line

The simplest approach is to run commands in a Docker container that has the Java command line tools pre-installed.

First, build and start the container:

```
docker-compose -p redisconf-2019 up -d --build java-development
```

This creates a container that has mounted the current directory containing the source code.

Second, run the command:

```
docker exec -it redisconf-2019_java-development_1 bash
```

You should now have a Linux prompt.

1.3.2. Less easy: Local command line

Alternatively, you can run commands locally on your laptop. To do that you need to install:

- Java 8/10
- curl (ideally)

Note: if you are running Windows, it is best that you use a Linux/bash-style command line such as:

- Windows Subsystem for Linux
- Git bash shell

You also need to set the `DOCKER_HOST_IP` environment variable to the **IP address** of your machine (NOT `127.0.0.1`).

You can verify that `DOCKER_HOST_IP` is set correctly by running this command:

```
./verify-docker-host-ip.sh
```

1.4. Compile the source code and start the services

Now that you have a command line (container-based or local) you can run commands.

Build and start the services:

```
./gradlew composeUp testClasses
```

This will

1. Download Gradle
2. Build the example application including test classes
3. Download the necessary Docker images
4. Start the services including MySql, Redis, and the Eventuate Tram CDC service

Note:

- This will ensure that the Java dependencies and Docker images are downloaded before the class.
- If you are running in a corporation that requires using an HTTP(S) proxy then will get an error such as 'Could not resolve all dependencies for configuration'. You will need to configure Gradle to use the proxy.

2. Setting up your IDE

If you are using a Java IDE, you need to import the code.

2.1. Importing the source code into your IDE

You need to import the project into your IDE:

- IntelliJ IDEA - `File→Open` the top-level `build.gradle` file. You will find it more convenient to enable auto-import.
- Eclipse - `File→Import→existing gradle project` and select the root directory You might need to install the Eclipse BuildShip plugin.

2.2. Setting environment variables in your IDE (Optional)

In order to run some of the tests within your IDE, you might need to set `DOCKER_HOST_IP`. There are a few different to do this:

1. Operating system-specific mechanism to set it for all processes
2. IDE specific mechanism

3. Set in `DOCKER_HOST_IP` command line/shell and launch IDE from the command line.

2.3. Refreshing your IDE

When you change a `build.gradle` file you will need to 'refresh' your IDE:

- IntelliJ IDEA - Go to the Gradle view and refresh. Note: if auto-import is enabled, IntelliJ IDEA will **often** pick up the changes automatically.
- Eclipse - In the project view, click right (on a file) and select `Gradle→Refresh`

2.4. Stopping the containers

To cleanup, run:

```
./gradlew composeDown
```

If you are running the `java-development` container then first exit the container and then run:

```
docker-compose -p redisconf-2019 down
```

You are now ready for the labs.

2.5. Cleaning up

Stopping the containers won’t delete what was downloaded. To do that:

1. Delete the images using `docker rmi`. Alternatively, you can use `docker-compose -p redisconf-2019 down --rmi all`, which will delete the locally built images and those that were downloaded **including redis**.
2. If you built locally, you might want to cleanup the `~/.gradle` directory.

3. Stories

This application implements the following stories.

3.1. Story: Register Customer

```
As a Customer
I want to Register
So that I can use place orders
```

3.2. Story: Place Order

```
As a Customer
I want to place an Order
So that I can obtain products
```

3.3. Story: View Order History

```
As a Customer
I want to view my order history
So that I can see the status of my orders
```

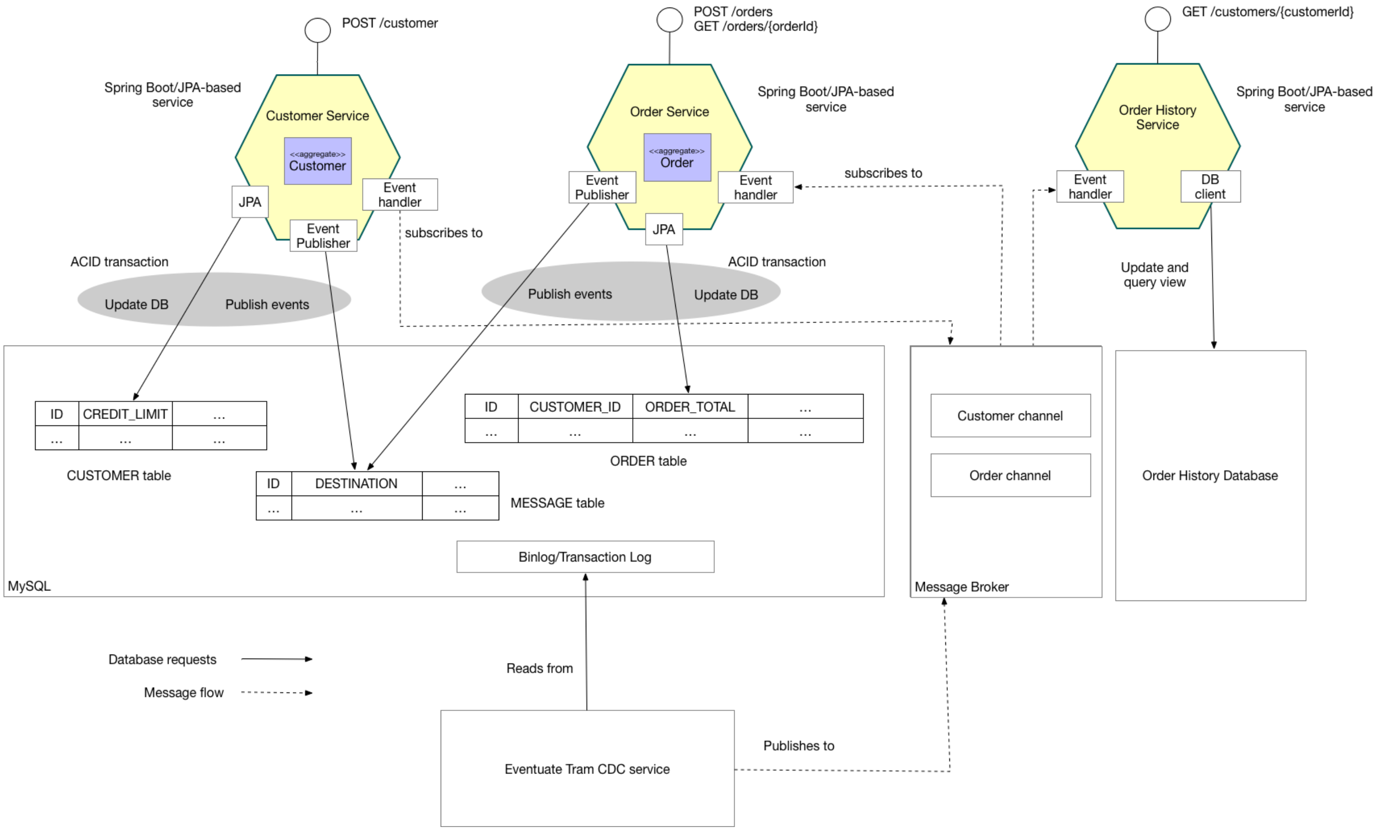
4. Order Management Domain model

The domain model for the Order management application consists of the following aggregates (a.k.a. business objects):

- `Customer` - a customer
- `Order` - an order placed by a customer

5. Application architecture

The following diagram shows the application architecture:



ID	CREDIT_LIMIT	...
...	...	...

ID	CUSTOMER_ID	ORDER_TOTAL	...
...	...	...	...

ID	DESTINATION	...
...	...	...

The system consists of the following application services:

- `Customer service`
  - responsible for managing customers
  - provides an API for creating customers
- `Order Service`
  - responsible for managing orders
  - provides an API for creating orders
- `Order History Service`
  - subscribes to events published by other services and updates a CQRS view in Redis.
  - provides an API for retrieving a customer, and their orders.

In addition, there are also the following infrastructure services:

- `Message Broker` - implemented by Redis Streams
- `Eventuate CDC service` - reads events/messages inserted into MySQL and publishes them to the Message Broker

- `Order History Database` - implemented using Redis

The application uses sagas to maintain data consistency across services. The sagas are implemented using Eventuate Tram Core.

6. Lab: Implement choreography-based saga in `Order Service`

The goal of this lab is to review the `Order Service`'s choreography-based `Create Order` saga.

The saga works as follows:

1. `Order Service` creates an `Order` in the `PENDING_CUSTOMER_VALIDATION` state
2. `Order Service` publishes an `OrderCreatedEvent`
3. `Customer Service` consumes the `OrderCreatedEvent` and attempts to reserve credit for the `Order`
4. `Customer Service` publishes either a `CustomerCreditReservedEvent` or a `CustomerCreditReservationFailedEvent`
5. `Order Service` consumes the `CustomerCreditReservedEvent` or a `CustomerCreditReservationFailedEvent` and changes the state of the `Order` to either `APPROVED` or `REJECTED`

There are following parts to this lab:

1. Review the code in the `Order Service` that publishes an `OrderCreatedEvent`
2. Review the code in the `Customer Service`, which consumes this event, attempts to reserve credit and publishes either a `CustomerCreditReservedEvent` or a `CustomerCreditReservationFailedEvent`
3. Review the code in the `Order Service` that subscribes to the events published by the `Customer Service` and approves or rejects the `Order`.

6.1. Review the `OrderService` *class*

The `OrderService` class is responsible for publishing events when orders are created, approved and rejected. Take a look at the file

```
 ./order-
service/src/main/java/io/eventuate/examples/tram/ordersandcustomers/orders/servi
ce/OrderService.java
```

The `publishOrderCreatedEvent()` publishes the `OrderCreatedEvent`.

6.2. Review the event handler in the `Customer Service`

The `Customer Service` consumes the `OrderCreatedEvent` and attempts to reserve credit for the `Order` It then publishes either a `CustomerCreditReservedEvent` or a `CustomerCreditReservationFailedEvent`. The code that implements the behavior is the `CustomerServiceEventSubscriber` class.

Take a look at the file:

```
./customer-
service/src/main/java/io/eventuate/examples/tram/ordersandcustomers/customers/se
rvice/CustomerServiceEventSubscriber.java
```

The `handleOrderCreatedEvent()` method consumes an `OrderCreatedEvent` and attempts to reserve credit.

6.3. Review the event handler in the `Order Service`

The `OrderServiceEventSubscriber` class consumes events published by the `Customer Service`. For example, the `handleCustomerCreditReservedEvent()` method consumes a `CustomerCreditReservedEvent` and approves the `Order`.

Take a look at the file:

```
./order-
service/src/main/java/io/eventuate/examples/tram/ordersandcustomers/orders/servi
ce/OrderServiceEventSubscriber.java
```

6.4. Build the services and restart the Docker containers

To build the service and restart the modified `Order Service`, please run the following command:

```
./gradlew composeUp
```

6.5. Use the Swagger UI to create customers and orders

6.5.1. Create a customer

In your browser, visit `http://${DOCKER_HOST_IP}:8081/swagger-ui.html` to create a `Customer`.

Note: Replace `${DOCKER_HOST_IP}` with the IP address/hostname of where Docker is running. On a Mac, `localhost` works in URLs.

6.5.2. Create Orders

In your browser, visit `http://${DOCKER_HOST_IP}:8082/swagger-ui.html` to create an `Order`

6.6. Inspect the database

Look at the tables in MySQL:

```
$ ./mysql-cli.sh

mysql> show tables;
+-------------------------------+
| Tables_in_eventuate           |
+-------------------------------+
| customer                      |
| customer_credit_reservations |
| orders                        |
| message                       |
....

mysql> select * from customer;
+----+----------+--------+---------+
| id | amount   | name   | version |
+----+----------+--------+---------+
|  1 | 50000.00 | string |       1 |
+----+----------+--------+---------+
1 row in set (0.01 sec)
```

```
mysql> select * from orders;
+----+-------------+--------+-------+
| id | customer_id | amount | state |
+----+-------------+--------+-------+
|  1 |           1 |   1.00 |     1 |
+----+-------------+--------+-------+
1 row in set (0.00 sec)


mysql> select * from message\G
*************************** 1. row ***************************
           id: 000001699ce2eae8-0242ac1300070000
  destination:
io.eventuate.examples.tram.ordersandcustomers.customers.domain.Customer
      headers: {"PARTITION_ID":"1","event-aggregate-
type":"io.eventuate.examples.tram.ordersandcustomers.customers.domain.Customer",
"DATE":"Wed, 20 Mar 2019 20:55:10 GMT","event-aggregate-id":"1","X-B3-
SpanId":"b42e1670f01feaf0","event-
type":"io.eventuate.examples.tram.ordersandcustomers.customerservice.domain.even
ts.CustomerCreatedEvent","DESTINATION":"io.eventuate.examples.tram.ordersandcust
omers.customers.domain.Customer","X-B3-ParentSpanId":"0017977892c5157e","X-B3-
Sampled":"1","X-B3-TraceId":"0017977892c5157e","ID":"000001699ce2eae8-
0242ac1300070000"}
      payload: {"name":"string","creditLimit":{"amount":50000}}
    published: 0
creation_time: 1553115310841
*************************** 2. row ***************************
           id: 000001699ce309d4-0242ac1300080000
  destination: io.eventuate.examples.tram.ordersandcustomers.orders.domain.Order
      headers: {"PARTITION_ID":"1","event-aggregate-
type":"io.eventuate.examples.tram.ordersandcustomers.orders.domain.Order","DATE"
:"Wed, 20 Mar 2019 20:55:18 GMT","event-aggregate-id":"1","X-B3-
SpanId":"79e311877e999f13","event-
type":"io.eventuate.examples.tram.ordersandcustomers.orderservice.domain.events.
OrderCreatedEvent","DESTINATION":"io.eventuate.examples.tram.ordersandcustomers.
orders.domain.Order","X-B3-ParentSpanId":"60c0d8b0385a3b26","X-B3-
Sampled":"1","X-B3-TraceId":"60c0d8b0385a3b26","ID":"000001699ce309d4-
0242ac1300080000"}
      payload: {"orderDetails":{"customerId":1,"orderTotal":{"amount":1}}}
    published: 0
creation_time: 1553115318747
*************************** 3. row ***************************
```

```
           id: 000001699ce30bc6-0242ac1300070000
  destination:
io.eventuate.examples.tram.ordersandcustomers.customers.domain.Customer
      headers: {"PARTITION_ID":"1","event-aggregate-
type":"io.eventuate.examples.tram.ordersandcustomers.customers.domain.Customer",
"DATE":"Wed, 20 Mar 2019 20:55:19 GMT","event-aggregate-id":"1","X-B3-
SpanId":"c256628612375b9f","event-
type":"io.eventuate.examples.tram.ordersandcustomers.customerservice.domain.even
ts.CustomerCreditReservedEvent","DESTINATION":"io.eventuate.examples.tram.orders
andcustomers.customers.domain.Customer","X-B3-
ParentSpanId":"4d7adbc8e8bf241b","X-B3-Sampled":"1","X-B3-
TraceId":"60c0d8b0385a3b26","ID":"000001699ce30bc6-0242ac1300070000"}
      payload: {"orderId":1}
    published: 0
creation_time: 1553115319240
*************************** 4. row ***************************
           id: 000001699ce30c15-0242ac1300080000
  destination: io.eventuate.examples.tram.ordersandcustomers.orders.domain.Order
      headers: {"PARTITION_ID":"1","event-aggregate-
type":"io.eventuate.examples.tram.ordersandcustomers.orders.domain.Order","DATE"
:"Wed, 20 Mar 2019 20:55:19 GMT","event-aggregate-id":"1","X-B3-
SpanId":"d0a1812e3c09fc18","event-
type":"io.eventuate.examples.tram.ordersandcustomers.orderservice.domain.events.
OrderApprovedEvent","DESTINATION":"io.eventuate.examples.tram.ordersandcustomers
.orders.domain.Order","X-B3-ParentSpanId":"e194ff19dcc2a6b4","X-B3-
Sampled":"1","X-B3-TraceId":"60c0d8b0385a3b26","ID":"000001699ce30c15-
0242ac1300080000"}
      payload: {"orderDetails":{"customerId":1,"orderTotal":{"amount":1.00}}}
    published: 0
creation_time: 1553115319319
4 rows in set (0.01 sec)
mysql> quit
Bye
```

6.7. Inspect the interaction between the services

This application is configured to use distributed tracing. You can view the traces in the Zipkin console:

1. Open the web page `http://${DOCKER_HOST_IP}:9411`

2. Click the `Find Traces Button`

You should see a trace for each time you created a customer or order.

6.8. Inspect Redis

You can also look at the Redis streams created by the application:

```
$ ./redis-cli.sh
192.168.1.109:6379> keys *
 1) "eventuate-tram:redis:group:member:consumerServiceEvents:3fdbd11e-5599-457b-
a41c-94e9f6e6e208"
...

192.168.1.109:6379> type
'io.eventuate.examples.tram.ordersandcustomers.customers.domain.Customer-1'
stream

192.168.1.109:6379> XINFO STREAM
io.eventuate.examples.tram.ordersandcustomers.orders.domain.Order-1

...

192.168.1.109:6379> XINFO GROUPS
io.eventuate.examples.tram.ordersandcustomers.orders.domain.Order-1
1) 1) "name"
   2) "customerServiceEventSubscriber"
   3) "consumers"
   4) (integer) 1
   5) "pending"
   6) (integer) 0
   7) "last-delivered-id"
   8) "1553115319339-0"
...

192.168.1.109:6379> XINFO consumers
io.eventuate.examples.tram.ordersandcustomers.orders.domain.Order-1
orderServiceEvents
1) 1) "name"
   2) "customerServiceEventSubscriber"
   3) "pending"
   4) (integer) 0
   5) "idle"
   6) (integer) 1011036

192.168.1.109:6379> quit
```

7. Lab: Implement a CQRS view using Redis

In this lab, you will implement the `View Order History` story:

```
As a Customer
I want to view my order history
So that I can see the status of my orders
```

The `Order History Service` that subscribes to `Customer`, and `Order` events and maintains a CQRS view in Redis It creates a key-value pair of type `HASH` in Redis for each `Customer`. This `HASH` contains the customer's information, and their `Orders`. The `Order History Service` provides a REST endpoint - `GET /customers/{customerId}` - for retrieving a customer's order history.

7.1. Review the event handling code

The `OrderHistoryServiceEventSubscriber` class is responsible for subscribing to events and updating Redis. It defines several event handlers. Each method invokes a method on `OrderHistoryViewService` to update Redis. The `OrderHistoryViewService` class updates Redis using Spring Data for Redis.

Take a look at the file:

```
./order-history-
service/src/main/java/io/eventuate/examples/tram/ordersandcustomers/orderhistory
service/service/OrderHistoryViewService.java
```

7.2. Use the Swagger UI

You can use the Swagger UI - `http://${DOCKER_HOST_IP}:8083/swagger-ui.html` - to retrieve a customer's order history.

7.3. Look at the database

Use the following commands to look at the CQRS view in Redis.

```
$ ./redis-cli.sh
192.168.1.109:6379> keys
io.eventuate.examples.tram.ordersandcustomers.orderhistory*
 1)
"io.eventuate.examples.tram.ordersandcustomers.orderhistory.common.CustomerView:
10"
 2)
"io.eventuate.examples.tram.ordersandcustomers.orderhistory.common.CustomerView:
19"
 3)
"io.eventuate.examples.tram.ordersandcustomers.orderhistory.common.OrderView:6"
 4)
"io.eventuate.examples.tram.ordersandcustomers.orderhistory.common.OrderView:8"
 5)
"io.eventuate.examples.tram.ordersandcustomers.orderhistory.common.OrderView:12"
 6)
"io.eventuate.examples.tram.ordersandcustomers.orderhistory.common.OrderView:9"

192.168.1.109:6379> type
"io.eventuate.examples.tram.ordersandcustomers.orderhistory.common.CustomerView:
10"
hash

192.168.1.109:6379> HGETALL
"io.eventuate.examples.tram.ordersandcustomers.orderhistory.common.CustomerView:
10"
1) "id"
2) "10"
3) "name"
4) "John"
5) "creditLimit.amount"
6) "1000"
7) "orders.[14].state"
8) "APPROVED"
9) "orders.[14].orderId"
10) "14"
11) "orders.[14].orderTotal.amount"
12) "100"
```

```
13) "orders.[15].state"
14) "REJECTED"
15) "orders.[15].orderId"
16) "15"
17) "orders.[15].orderTotal.amount"
18) "1000"
```