Developer Solutions

Develop Microservices application using CQRS and Event sourcing patterns in Oracle Cloud

Posted by Venkatesh Yadavalli-Oracle in Developer Solutions on Mar 30, 2017 10:08:02 AM

This blog introduces you to building event-driven microservices application using CQRS and Event sourcing patterns. Following is a brief definition of the concepts that would be discussed during the course of this blog, more details about these can be obtained from the resources provided at the end of this blog.

What is a Microservice?
While there is no single definition for this architectural style, Adrian Cockcroft defines microservices architecture as a service-oriented architecture composed of loosely coupled elements that have bounded contexts.

What is a Bounded Context?
A Bounded Context is a concept that encapsulates the details of a single domain, such as domain model, data model, application services, etc., and defines the integration points with other bounded contexts/domains.

What is CQRS?
Command Query Responsibility Segregation (CQRS) is an architectural pattern that segregates the domain operations into two categories – Queries and Commands. While queries just return some results without making any state changes, commands are the operations which change the state of the domain model.

Why CQRS?
During the lifecycle of an application, it is common that the logical model becomes more complicated and structured that could impact the user experience, which must be independent of the core system.
In order to have a scalable and easy to maintain application we need to reduce the constraints between the read model and the write model. Some reasons for keeping reads and writes apart could be:

- Scalability (read exceeds the write, so does the scaling requirements for each differs and can be addressed better)
- Flexibility (separate read / write models)
- Reduced Complexity (shifting complexity into separate concerns)

What is Event sourcing?
Event sourcing achieves atomicity by using a different, event-centric approach to persisting business entities. Instead of storing the current state of an entity, the application stores a sequence of 'events' that changed the entity's state. The application can reconstruct an entity's current state by replaying the events. Since saving an event is a single operation, it is inherently atomic and does not require 2PC (2-phase commit) which is typically associated with distributed transactions.

Overview
This blog explains how CQRS and Event Sourcing patterns can be applied to develop a simple microservice application that consists of a single bounded context called "Cart" with add, remove and read operations. The sample does not have any functional significance but should be good enough to understand the underlying patterns and their implementations. The following diagram depicts a high level flow of activities when using CQRS and Event sourcing patterns to build applications:

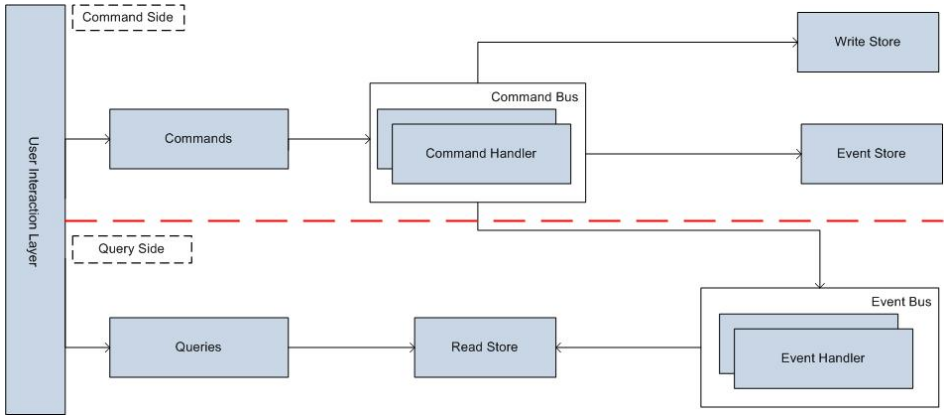

Figure 1 CQRS and Event sourcing

The sample referred in this blog uses the following technology stack:

- Spring Boot for building and packaging the application
- Axon framework with Spring for CQRS and Event Sourcing. Axon is an open source CQRS framework for Java which provides implementations of the most important building blocks, such as aggregates, repositories and event buses that help us build applications using CQRS and Event sourcing architectural patterns. It also allows you to provide your own implementation of the above mentioned building blocks.
- Oracle Application Container cloud for application deployment

With this background, let us start building the sample.

Identify Aggregate Root

First step is to identify the bounded context and domain entities in the bounded context. This will help us define the Aggregate Root (for example, an 'account', an 'order'…etc.). An aggregate is an entity or group of entities that is always kept in a consistent state. The aggregate root is the object on top of the aggregate tree that is responsible for maintaining this consistent state.
To keep things simple for this blog, we consider 'Cart' as the only Aggregate Root in the domain model. Just like the usual shopping cart, the items in the cart are adjusted based on the additions or removals happening on that cart.

Define Commands

This aggregate root has 2 commands associated with it:

- Add to Cart Command – Modeled by AddToCartCommand class
- Remove from Cart Command – Modeled by RemoveFromCartCommand class

```
01. publicclass AddToCartCommand {
02.
03.     private final String cartId;
04.     private final int item;
05.
06.     public AddToCartCommand(String cartId, int item) {
07.         this.cartId = cartId;
08.         this.item = item;
09.     }
10.
11.     public String getCartId() {
12.         return cartId;
13.     }
```

```
14.
15.     public int getItem() {
16.         return item;
17.     }
18. }
19.
20. public class RemoveFromCartCommand {
21.
22.  private final String cartId;
23.  private final int item;
24.
25.  public RemoveFromCartCommand(String cartId, int item) {
26.       this.cartId = cartId;
27.       this.item = item;
28.  }
29.
30.  public String getCartId() {
31.       return cartId;
32.  }
33.
34.  public int getItem() {
35.       return item;
36.  }
37.   }
```

As you notice, these commands are just POJOs used to capture the intent of what needs to happen within a system along with the necessary information that is required. Axon Framework does not require commands to implement any interface nor extend any class.

Define Command Handlers

A command is intended to have only one handler, the following classes represent the handlers for Add to Cart and Remove from Cart commands:

```
01. @Component
02. public class AddToCartCommandHandler {
03.
04.  private Repository repository;
05.
06.  @Autowired
07.  public AddToCartCommandHandler(Repository repository) {
08.       this.repository = repository;
09.  }
10.
11.  @CommandHandler
12.  public void handle(AddToCartCommand addToCartCommand){
13.       Cart cartToBeAdded = (Cart) repository.load(addToCartCommand.getCartId());
14.       cartToBeAdded.addCart(addToCartCommand.getItem());
15.  }
16.
17. }
18.
19. @Component
20. public class RemoveFromCartHandler {
21.
22.  private Repository repository;
23.
24.  @Autowired
25.  public RemoveFromCartHandler(Repository repository) {
26.       this.repository = repository;
27.     }
28.
```

```
 @CommandHandler
 public void handle(RemoveFromCartCommand removeFromCartCommand){
      Cart cartToBeRemoved = (Cart) repository.load(removeFromCartCommand.getCartId());
      cartToBeRemoved.removeCart(removeFromCartCommand.getItem());

 }
}
```

We use Axon with Spring framework, so the Spring beans defined above have methods annotated with @CommandHandler which makes them as command handlers. @Component annotation ensures that these beans are scanned during application startup and any auto wired resources are injected into this bean. Instead of accessing the Aggregates directly, Repository which is a domain object in Axon framework abstracts retrieving and persisting of aggregates.

Application Startup

Following is the AppConfiguration class which is a Spring configuration class that gets initialized upon application deployment and creates the components required for implementing the patterns.

```
@Configuration
@AnnotationDriven
public class AppConfiguration {

 @Bean
 public DataSource dataSource() {
      return DataSourceBuilder
                .create()
                .username("sa")
                .password("")
                .url("jdbc:h2:mem:axonappdb")
                .driverClassName("org.h2.Driver")
                .build();
 }

 /**
 * Event store to store events
 */
 @Bean
 public EventStore jdbcEventStore() {
      return new JdbcEventStore(dataSource());
 }

 @Bean
 public SimpleCommandBus commandBus() {
      SimpleCommandBus simpleCommandBus = new SimpleCommandBus();
      return simpleCommandBus;
 }

 /**
 *  Cluster event handlers that listens to events thrown in the application.
 */
 @Bean
 public Cluster normalCluster() {
      SimpleCluster simpleCluster = new SimpleCluster("simpleCluster");
      return simpleCluster;
 }


 /**
 * This configuration registers event handlers with defined clusters
 */
 @Bean
```

```
 public ClusterSelector clusterSelector() {
      Map<String, Cluster> clusterMap = new HashMap<>();
      clusterMap.put("msacqrses.eventhandler", normalCluster());
      return new ClassNamePrefixClusterSelector(clusterMap);
 }

 /**
*The clustering event bus is needed to route events to event handlers in the clusters.
 */
 @Bean
 public EventBus clusteringEventBus() {
     ClusteringEventBus clusteringEventBus = new ClusteringEventBus(clusterSelector(), terminal());

     return clusteringEventBus;
 }

 /**
 * Event Bus Terminal publishes domain events to the cluster
 *
 */
 @Bean
 public EventBusTerminal terminal() {
      return new EventBusTerminal() {
            @Override
            public void publish(EventMessage... events) {
                normalCluster().publish(events);
 }
            @Override
            public void onClusterCreated(Cluster cluster) {

            }
 };
 }

 /**
 * Command gateway through which all commands in the application are submitted
 *
 */

 @Bean
 public DefaultCommandGateway commandGateway() {
      return new DefaultCommandGateway(commandBus());
 }

 /**
* Event Repository that handles retrieving of entity from the stream of events.
 */
 @Bean
 public Repository<Cart> eventSourcingRepository() {
EventSourcingRepository eventSourcingRepository = new EventSourcingRepository(Cart.class, jdbcEventStore());
      eventSourcingRepository.setEventBus(clusteringEventBus());

     return eventSourcingRepository;
 }
}
```

Let us take a look at the key Axon provided infrastructure components that are initialized in this class:

Command bus

As represented in “Figure 1” above, command bus is the component that routes commands to their respective command handlers. Axon Framework comes with different types of Command Bus out of the box that can be used to dispatch commands to command handlers. Please refer here for more details on Axon’s Command Bus implementations. In our example, we use *SimpleCommandBus* which is configured as a bean in Spring's application context.

Command Gateway

Command bus can directly send commands but it is usually recommended to use a command gateway. Using a command gateway allows developers to perform certain functionalities like intercepting commands, setting retry in failure scenarios…etc. In our example, we use Axon provided default which is *DefaultCommandGateway* that is configured as a Spring bean to send commands instead of directly using a command bus.

Event Bus

As depicted in “Figure 1”, the commands initiated on an Aggregate root are sent as events to the Event store where they get persisted. Event Bus is the infrastructure that routes events to event handlers. Event Bus may look similar to command bus from a message dispatching perspective but they vary fundamentally.
Command Bus works with commands that define what happen in the near future and there is only one command handler that interprets the command. However in case of Event Bus, it routes events and events define actions that happened in the past with zero or more event handlers for an event.
Axon defines multiple implementations of Event Bus, in our example we use *ClusteringEventBus* which is again wired up as a Spring bean. Please refer here for more details on Axon’s Event Bus implementations.

Event Store

We need to configure an event store as our repository will store domain events instead of the current state of our domain objects. Axon framework allows storing the events using multiple persistent mechanisms like JDBC, JPA, file system etc. In this example we use a JDBC event store.

Event Sourcing Repository

In our example, the aggregate root is not created from a representation in a persistent mechanism, instead is created from stream of events which is achieved through an Event sourcing repository. We configure the repository with the event bus that we defined earlier since it will be publishing the domain events.

Database

We use in memory database (h2) in our example as the data store. The Spring Boot’s application.properties contains the data source configuration settings:

```
# Datasource configuration
spring.datasource.url=jdbc:h2:mem:axonappdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.validation-query=SELECT 1;
spring.datasource.initial-size=2
spring.datasource.sql-script-encoding=UTF-8

spring.jpa.database=h2
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=create
```

As mentioned above, this example uses a JDBC event store to store the domain events generated in the system. These events are stored in a default tables (part of Axon framework event infrastructure) specified by the Axon framework. We use the following startup class for creating the database tables required by this example:

```
@Component
public class Datastore {

 @Autowired
 @Qualifier("transactionManager")
 protected PlatformTransactionManager txManager;

 @Autowired
```

```
09.  private Repository repository;
10.
11.  @Autowired
12.  private javax.sql.DataSource dataSource;
13.     // create two cart entries in the repository used for command processing
14.  @PostConstruct
15.  private void init(){
16.
17.  TransactionTemplate transactionTmp = new TransactionTemplate(txManager);
18.  transactionTmp.execute(new TransactionCallbackWithoutResult() {
19.             @Override
20.             protected void doInTransactionWithoutResult(TransactionStatus status) {
21.                 UnitOfWork uow = DefaultUnitOfWork.startAndGet();
22.                 repository.add(new Cart("cart1"));
23.                 repository.add(new Cart("cart2"));
24.                 uow.commit();
25.             }
26.  });
27.
28.  // create a database table for querying and add two cart entries
29.  JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
30.  jdbcTemplate.execute("create table cartview (cartid VARCHAR , items NUMBER )");
31.  jdbcTemplate.update("insert into cartview (cartid, items) values (?, ?)", new Object[]{"cart1", 0});
32.  jdbcTemplate.update("insert into cartview (cartid, items) values (?, ?)", new Object[]{"cart2", 0});
33.  }
```

This startup class creates two cart entries in the repository used for command processing and creates a database table called “cartview” which is used for processing the queries.

A quick recap on what we did so far:

- We have identified “Cart” as our Aggregate root and have defined commands and command handlers for adding and removing items from the Cart.
- We have defined a startup class which initializes the infrastructure components required for CQRS and Event sourcing.
- A startup class has also been defined to create the database tables and setup the data required by this sample.

Let us now look at our AggregateRoot - “Cart” which is defined as below:

Aggregate Root

```
01.  publicclass Cart extends AbstractAnnotatedAggregateRoot {
02.   @AggregateIdentifier
03.   private String cartid;
04.
05.   private int items;
06.
07.   public Cart() {
08.   }
09.
10.   public Cart(String cartId) {
11.        apply(new CartCreatedEvent(cartId));
12.   }
13.
14.   @EventSourcingHandler
15.   public void applyCartCreation(CartCreatedEvent event) {
16.        this.cartid = event.getCartId();
17.        this.items = 0;
18.   }
19.
```

```
 public void removeCart(int removeitem) {

 /**
* State is not directly changed, we instead apply event that specifies what happened. Events applied are stored.
*/
 if(this.items > removeitem && removeitem > 0)
      apply(new RemoveFromCartEvent(this.cartid, removeitem, this.items));

 }



 @EventSourcingHandler
 private void applyCartRemove(RemoveFromCartEvent event) {
 /**
* When events stored in the event store are applied on an Entity this method is
* called. Once all the events in the event store are applied, it will bring the Cart * to the most recent state.
*/

      this.items -= event.getItemsRemoved();
 }

 public void addCart(int item) {
 /**
* State is not directly changed, we instead apply event that specifies what happened. Events applied are stored.
*/
 if(item > 0)
      apply(new AddToCartEvent(this.cartid, item, this.items));
 }

 @EventSourcingHandler
 private void applyCartAdd(AddToCartEvent event) {
 /**
* When events stored in the event store are applied on an Entity this method is
* called. Once all the events in the event store are applied, it will bring the
* Cart to the most recent state.
*/

      this.items += event.getItemAdded();
 }

 public int getItems() {
      return items;
 }

 public void setIdentifier(String id) {
      this.cartid = id;
 }

 @Override
 public Object getIdentifier() {
      return cartid;
 }
}
```

Following are some key aspects of the above Aggregate Root definition:

1. The @AggregateIdentifier is similar to @Id in JPA which marks the field that represents the entity's identity.
2. Domain driven design recommends domain entities to contain relevant business logic, hence the business methods in the above definition. Please refer to the "References" section for more details.

3. When a command gets triggered, the domain object is retrieved from the repository and the respective method (say addCart) is invoked on that domain object (in this case “Cart”).
    a. The domain object instead of changing the state directly, applies the appropriate event.
    b. The event is stored in the event store and the respective handler gets triggered which makes the change to the domain object.
4. Note that the “Cart” aggregate root is only used for updates (i.e. state change via commands). All the query requests are handled by a different database entity (will be discussed in coming sections).

Let us also look at the events and the event handlers that manage the domain events triggered from “Cart” entity:

Events

As mentioned in the previous sections, there are two commands that get triggered on the “Cart” entity – Add to Cart and Remove from Cart. These commands when executed on the Aggregate root will generate two events – AddToCartEvent and RemoveFromCartEvent which are listed below:

```
01. publicclass AddToCartEvent {
02.
03.  private final String cartId;
04.  private final int itemAdded;
05.  private final int items;
06.  private final long timeStamp;
07.
08.  public AddToCartEvent(String cartId, int itemAdded, int items) {
09.       this.cartId = cartId;
10.       this.itemAdded = itemAdded;
11.       this.items = items;
12.       ZoneId zoneId = ZoneId.systemDefault();
13.       this.timeStamp = LocalDateTime.now().atZone(zoneId).toEpochSecond();
14.  }
15.
16.  public String getCartId() {
17.       return cartId;
18.  }
19.
20.  public int getItemAdded() {
21.       return itemAdded;
22.  }
23.
24.  public int getItems() {
25.       return items;
26.  }
27.
28.  public long getTimeStamp() {
29.       return timeStamp;
30.  }
31. }
32.
33. public class RemoveFromCartEvent {
34.  private final String cartId;
35.  private final int itemsRemoved;
36.  private final int items;
37.  private final long timeStamp;
38.
39.  public RemoveFromCartEvent(String cartId, int itemsRemoved, int items) {
40.       this.cartId = cartId;
41.       this.itemsRemoved = itemsRemoved;
42.       this.items = items;
43.       ZoneId zoneId = ZoneId.systemDefault();
```

```
        this.timeStamp = LocalDateTime.now().atZone(zoneId).toEpochSecond();

    }

    public String getCartId() {
        return cartId;
    }

    public int getItemsRemoved() {
        return itemsRemoved;
    }

    public int getItems() {
        return items;
    }

    public long getTimeStamp() {
        return timeStamp;
    }
}
```

Event Handlers

The events described above would be handled by the following event handlers:

```
@Component
public class AddToCartEventHandler {

    @Autowired
    DataSource dataSource;

    @EventHandler
    public void handleAddToCartEvent(AddToCartEvent event, Message msg) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        // Get current state from event
        String cartId = event.getCartId();
        int items = event.getItems();
        int itemToBeAdded = event.getItemAdded();
        int newItems = items + itemToBeAdded;


        //  Update cartview
        String updateQuery = "UPDATE cartview SET items = ? WHERE cartid = ?";
        jdbcTemplate.update(updateQuery, new Object[]{newItems, cartId});

    }
@Component
public class RemoveFromCartEventHandler {

    @Autowired
    DataSource dataSource;

    @EventHandler
    public void handleRemoveFromCartEvent(RemoveFromCartEvent event) {

        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        // Get current state from event
```

```
        String cartId = event.getCartId();
        int items = event.getItems();
        int itemsToBeRemoved = event.getItemsRemoved();
        int newItems = items - itemsToBeRemoved;

        // Update cartview
        String update = "UPDATE cartview SET items = ? WHERE cartid = ?";
        jdbcTemplate.update(update, new Object[]{newItems, cartId});

    }
}
```

As you notice, the event handlers update the “cartview” database table which is used for querying the “Cart” entity. While the commands get executed on one domain, the query requests are serviced by a different domain there by achieving CQRS with Event sourcing.

Controllers

This example defines 2 Spring controller classes one each for updating and querying the Cart domain. These are REST endpoints that could be invoked from a browser as below:

http://<host>:<port>/add/cart/<noOfItems >
http://<host>:<port>/remove/cart/<noOfItems >
http://<host>:<port>/view

```
@RestController
public class CommandController {

 @Autowired
 private CommandGateway commandGateway;

 @RequestMapping("/remove/{cartId}/{item}")
 @Transactional
 public ResponseEntity doRemove(@PathVariable String cartId, @PathVariable int item) {
       RemoveFromCartCommand removeCartCommand = new RemoveFromCartCommand(cartId, item);
      commandGateway.send(removeCartCommand);

      return new ResponseEntity<>("Remove event generated. Status: "+ HttpStatus.OK, HttpStatus.OK);
 }

 @RequestMapping("/add/{cartId}/{item}")
 @Transactional
 public ResponseEntity doAdd(@PathVariable String cartId, @PathVariable int item) {

      AddToCartCommand addCartCommand = new AddToCartCommand(cartId, item);
      commandGateway.send(addCartCommand);

     return new ResponseEntity<>("Add event generated. Status: "+ HttpStatus.OK, HttpStatus.OK);
 }


}

@RestController
public class ViewController {

 @Autowired
 private DataSource dataSource;

```

```
@RequestMapping(value = "/view", method = RequestMethod.GET, produces = MediaType.APPLICATION_JSON_VALUE)
 public ResponseEntity getItems() {

      JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
      List<Map<String, Integer>> queryResult = jdbcTemplate.query("SELECT * from cartview ORDER BY cartid", (rs, rowNum) -> {

      return new HashMap<String, Integer>() {{
                put(rs.getString("CARTID"), rs.getInt("ITEMS"));
             }};
 });

      if (queryResult.size() > 0) {
        return new ResponseEntity<>(queryResult, HttpStatus.OK);
      } else {
        return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
      }

 }

}
```

Deployment

We use Spring Boot to package and deploy the application as a runnable jar into Oracle Application Container cloud. The following Spring Boot class initializes the application:

```
@SpringBootApplication

public class AxonApp {

 // Get PORT and HOST from Environment or set default
 public static final Optional<String> host;
 public static final Optional<String> port;
 public static final Properties myProps = new Properties();

 static {
      host = Optional.ofNullable(System.getenv("HOSTNAME"));
      port = Optional.ofNullable(System.getenv("PORT"));
 }

 public static void main(String[] args) {
      // Set properties
      myProps.setProperty("server.address", host.orElse("localhost"));
      myProps.setProperty("server.port", port.orElse("8128"));

      SpringApplication app = new SpringApplication(AxonApp.class);
      app.setDefaultProperties(myProps);
      app.run(args);

 }
}
```

Create an xml file with the following content and place it in the same directory as the pom.xml. This specifies the deployment assembly of the application being deployed to Oracle Application Container Cloud.

```
<assembly xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3 http://maven.apache.org/xsd/assembly-1.1.3.xsd">
 <id>dist</id>
 <formats>
```

```
 <format>zip</format>
 </formats>
 <includeBaseDirectory>false</includeBaseDirectory>
 <files>
 <file>
            <source>manifest.json</source>
            <outputDirectory></outputDirectory>
 </file>
 </files>
 <fileSets>
 <fileSet>
            <directory>${project.build.directory}</directory>
            <outputDirectory></outputDirectory>
            <includes>
                <include>${project.artifactId}-${project.version}.jar</include>
            </includes>
 </fileSet>
 </fileSets>
</assembly>
```

To let Application Container cloud know the jar to run once the application is deployed, you need to create a “manifest.json” file specifying the jar name as shown below:

```
{
 "runtime": {
 "majorVersion": "8"
    },
 "command": "java -jar AxonApp-0.0.1-SNAPSHOT.jar",
 "release": {},
 "notes": "Axon Spring Boot App"
}
```

The following diagram depicts the project structure of this sample:

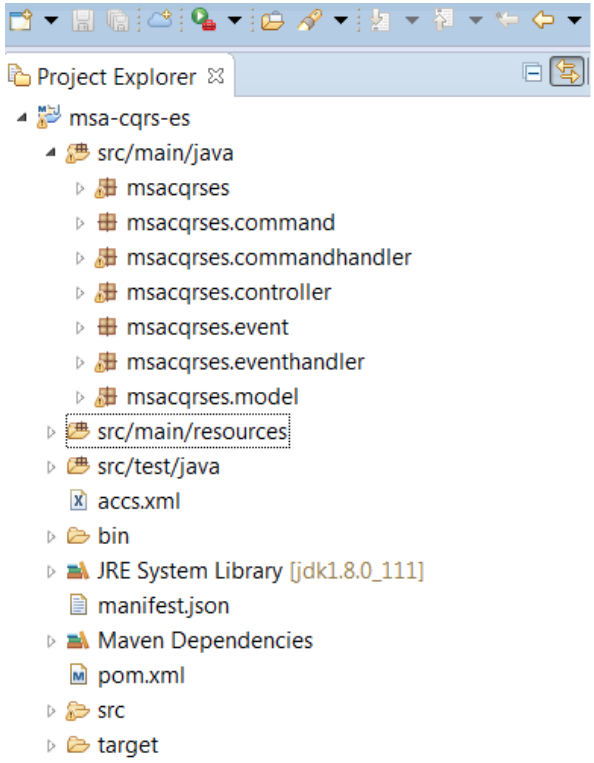

Figure 2 Project Structure

The application jar file along with the above manifest file should be archived to zip and uploaded into Application Container cloud for deployment. Please refer here for more details on deploying Spring Boot Application in Application Container cloud.

Once the application is successfully deployed, you would be able to access the following URLs to trigger the services on the Cart:
http://<host>:<port>/view
http://<host>:<port>/add/cart/<noOfItems >
http://<host>:<port>/remove/cart/<noOfItems >

When you first hit the “view” REST endpoint, you can see the 2 carts that we added in our startup class with number of items added to them. You can add or remove items from the Cart using the other two REST calls and can retrieve the updated item count using the “view” REST call. The result from the above REST invocations is a simple JSON structure displaying the Carts and the no of items in the Cart at a given point of time.

Conclusion

This blog is restricted to introduce you to developing microservice applications using CQRS and Event sourcing patterns. You can refer to the following resources to know more about other advanced concepts and recent updates in this space.

References

- Introduction to domain modeling, CQRS and Event Sourcing
- Domain Driven Design for Services Architecture
- Axon Framework
- Eventuate.io
- Developing for Oracle Application Container Cloud Service

The views expressed in this post are my own and do not necessarily reflect the views of Oracle.

903 Views Categories: PaaS, Design, Develop, Web Services
Tags: java, microservices, spring framework, database in memory, sourcing event, oracle application container cloud service, java spring, spring boot, rest apis, axon, cqrs

Average User Rating

(0 ratings)

0 Comments