

Build an application using microservices and CQRS

Maximize the performance and scalability of your microservices-based apps

By Felicia Tucci | Published July 27, 2016 - Updated July 26, 2016

[CloudContainersMicroservices](#)

Simply put, microservices technology is a style of breaking large software applications into smaller services, each of which performs a specific task. Although microservices technology is widely associated with performance and scalability, there's no guarantee that you will achieve those objectives by using microservices in an Internet-grade environment.

In order to fully exploit the microservices promise, the technology must be complemented by an appropriate architectural *pattern* that maximizes the potential benefits of the approach. Command Query Responsibility Segregation (CQRS) is one of those patterns, and probably the most relevant.

This article explores the benefits of developing microservices according to the CQRS pattern. I lead the reader through the development of an example application written in Java™, applying the Axon Framework, and then demonstrate the application's deployment on a cloud environment or on Docker containers.

[Get the code](#)

Regardless of the complexity of the business logic, with these techniques the implementation of new commands, events, and event listeners can become essentially a purely mechanical and repetitive process."

Microservices

In cloud ecosystems, microservices is not just a current architectural trend—it has become a requirement. Using the cloud as the standard support for the deployment of applications has inspired developers to rethink how apps are designed and developed. Microservices technology is an innovative new way to deal with the development of a cloud-native application. However, its architectural principles are applicable in contexts other than the cloud.

Some of the advantages of microservices include:

- Frequency of change
- Scalability
- Security
- Platform (for example, operating system)
- Infrastructure (such as RAM, network throughput)

CQRS pattern overview

Why should you choose the CQRS pattern to implement microservices? In traditional application architecture, applications generally act on data to be persisted in a database. A unique database is usually used for data model entities, accessed for both reading and writing. The design of the data is driven by the write and update operations to keep the data consistent. Developers try to minimize data redundancy using normalization techniques. While it's necessary to store data in normalized form, this can be detrimental to read operations. To extract some data, developers need to code complex SQL queries that join data from multiple tables.

Furthermore, in many applications, while data is created once and modified only occasionally, it may be read many, many times. Therefore, the developer needs to pay particular attention to reading performance; the data should be accessed by the smallest possible number of queries and the business logic executed during each query should be minimized. This is where the CQRS pattern comes in.

[Martin Fowler](#) of Thoughtworks says about CQRS: “At its heart is the notion that you can use a different model to update information than the model you use to read information.”

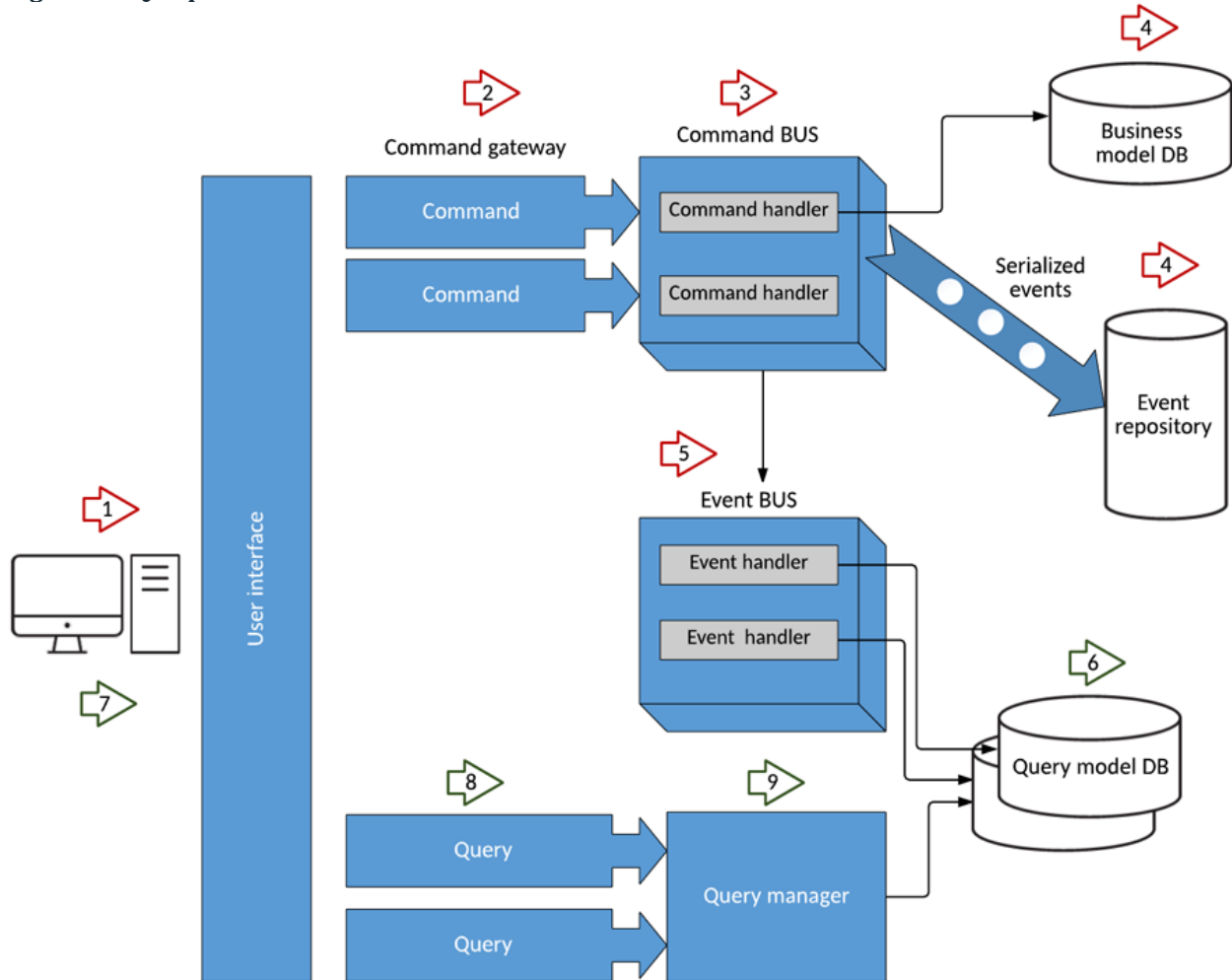
The CQRS pattern provides a guideline for identifying services and distributing various business aspects. The basic idea is to divide the operations that act on a domain object into two distinct categories:

- **Queries**—methods that return a result and do not change the system state.
- **Commands**—methods that change the system state but do not return values.

The good news is that queries and commands can be decoupled. The link between them is represented by the events generated by the command processing that feed the repositories for query data.

The events are persisted into the event repository. The business model database can therefore contain the last state of the system, but the event repository keeps the whole history of the data. With this separation of concerns, the developer is free to choose the most appropriate technology for each pattern component, and can (if desired) build a polyglot application. (For example, you could choose RDBMS for a database of commands, an in-memory database for an event repository, and NoSQL for a query database.)

Figure 1 illustrates the main pattern elements:
Figure 1. CQRS pattern



Axon Framework implementation of CQRS

A good Java implementation of the CQRS pattern is provided by [Axon Framework](#). This framework provides a simple programming model for implementing even complex applications, which frees up the developer to focus on the business logic (such as the definition of the commands and the events to generate and their handlers). With a strong framework configuration, you can define the types of command-bus, event-bus, and event-repository, as well as the transactional aspect between various components and the type of serialization for data communication. In my example, I use Spring to configure the framework. Note that in this case I work with only a very small subset of what the framework offers.

Step 1. Select the application context

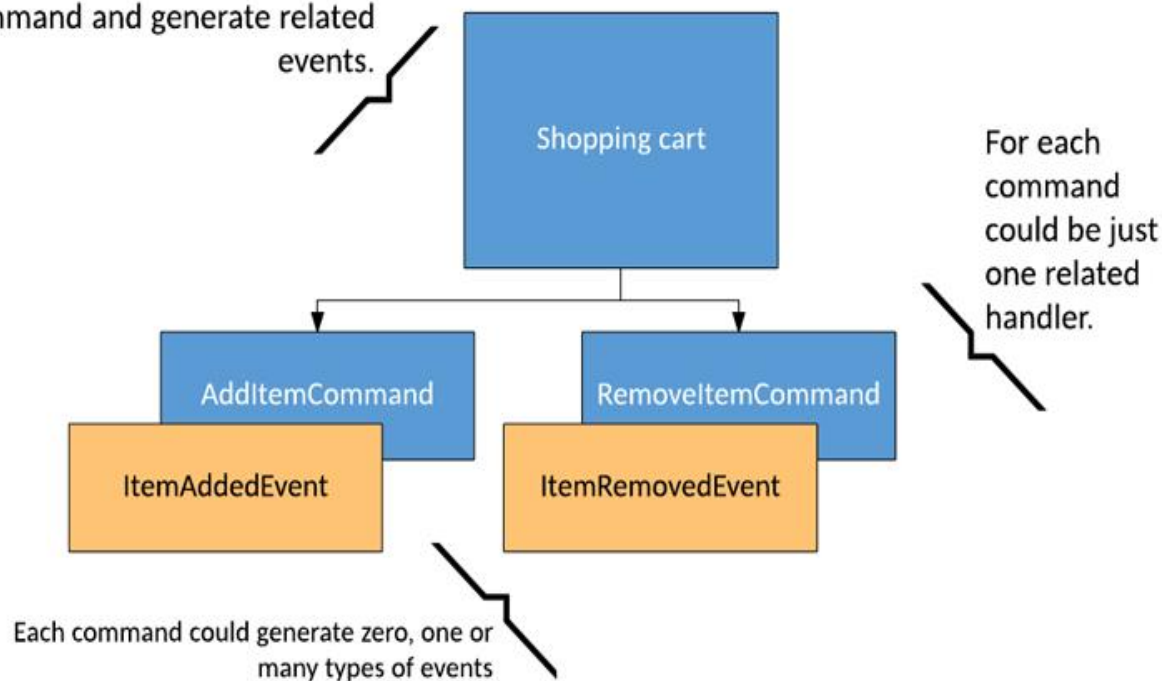
First, you need to define the bounded context and its domain entities. In that way, you will define your `AggregateRoot` (for example, an order, clinic visit, or

trip). According to the [Axon Framework website](#), 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.”

In my example, the bounded context is the (very simplified) cart of an online plant shop. As with any online shopping cart, a shopper using this cart can add items to it, remove them, empty the cart, view the items in the cart, and proceed to checkout when they're finished shopping.

Figure 2. ShoppingCart aggregate object

This is the AggregateRoot of DomainModel. It contains the commandHandlers to manipulate command and generate related events.



Note that an aggregate root is always recognizable by its `AggregateIdentifier`.

```
public class ShoppingCart extends AbstractAnnotatedAggregateRoot {
```

```
    @AggregateIdentifier
    private String id;
```

```
    public ShoppingCart() {
    }
```

```
    //.....
}
```

Into the Spring configuration, I link the aggregate root object (`ShoppingCart`) to its command bus and event repository:

```
<axon:aggregate-command-handler id="ShoppingCartHandler"
```

```
aggregate-type="cqrs.example.shoppingcart.model.ShoppingCart"
repository="shoppingCartRepository"
command-bus="commandBus" />
```

Step 2. Design commands and their handlers

Commands are objects that contain the data that changes the system state; they do not return results.

To manage the shopping cart, I have defined two commands:

- **AddItemCommand**—adds a certain number of an item (in this case, a plant) to the shopping cart. It is dispatched asynchronously.
- **RemoveItemCommand**—removes an object already in the cart, changing its availability in stock. It is dispatched asynchronously, because the system doesn't need to return any response to the user.

Command handlers are annotated methods contained in `ShoppingCart` (the `AggregateRoot`) class, which represents the main domain object. (The annotations allow you to use `AggregateRoot` as a simple Plain Old Java Object, or POJO.) The events are generated into the related command handler methods.

```
@CommandHandler
public ShoppingCart(AddItemCommand command) {
    id=command.getItemId();
    //put the business logic here
    //....
    apply(new ItemAddedEvent(command.getItemId()));
}

@CommandHandler
public ShoppingCart(RemoveItemCommand command) {
    id=command.getItemId();
    //put the business logic here
    //....
    apply(new ItemRemovedEvent(command.getItemId()));
}
```

Show more

In some cases, you might need to perform other actions (such as validation, logging, or authorization) regardless of the specific type of command. This is possible through the Command Handler Interceptors. These can take action both before and after processing a command. They can also completely block command processing. The interceptor must implement the interface `org.axonframework.commandhandling.CommandHandlerInterceptor`.

Step 3. Identify events and their listeners

As soon as the command handling is processed, the unit of work is committed and any action is completed. At this point, each repository is notified of status changes and the registered events are sent to the event bus for publication. If you

provide a listener to this event, it will be run asynchronously to the command and event generation.

In my simple application, two events can be generated:

- `ItemAddedEvent` (by `AddItemCommand`)
- `ItemRemovedEvent` (by `RemoveItemCommand`)

These events contain the data relevant to the event handling (for example, `itemId`, `price`, and `quantity`).

In the `ShoppingCartEventListener` class, I have registered a listener for each event.

```
public class ShoppingCartEventListener {

    @EventHandler
    public void onEvent(ItemAddedEvent event) {
        System.out.println("Received ItemAddedEvent id:" +
            event.getItemId() + " on thread named " +
                Thread.currentThread().getName());
    }

    @EventHandler
    public void onEvent(ItemRemovedEvent event) {
        System.out.println("Received ItemRemovedEvent id:" +
            event.getItemId() + " on thread named " +
                Thread.currentThread().getName());
    }

}
```

Show more

This is the right place to put the code for updating the data view on which the queries will act. Depending on the purpose of the listener, it is possible that the persistence of data is done on more than one database. Each database could have not only a separate model for queries, but also use a different technology (such as SQL DB, NoSQL DB, and so on).

Step 4. Configure the command handling

One of the most significant advantages of the Axon Framework is the ease of application configuration. In this section, I describe some of the configuration choices I made in my application.

Through the command bus, each command is dispatched to its respective handler. The command bus definition is inserted into the configuration Spring file.

```
<!-- Define a command bus -->
<axon:command-bus id="commandBus"
transaction-manager="transactionManager" />
```

In the `commandBus` definition, you'll see the transaction manager reference; it is used to manage the transaction during command handling.

```
<bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
    <property name="dataSource" ref="dataSource"/>
</bean>
```

In adopting the CQRS pattern for use in your application development, consider this *transactional* aspect of CQRS. Commands cannot be lost. You need a transaction manager (to handle **ACID** transactions) to ensure that every command is processed and that the events are generated and made persistent in the event store. This holds true for command handling, but if you consider the entire transaction (from running the command to the event listener execution) in regard to the asynchronous characteristics of flow, it is a **BASE** transaction. BASE is an acronym for **B**asic **A**vailable, **S**oft state, **E**ventually consistent.

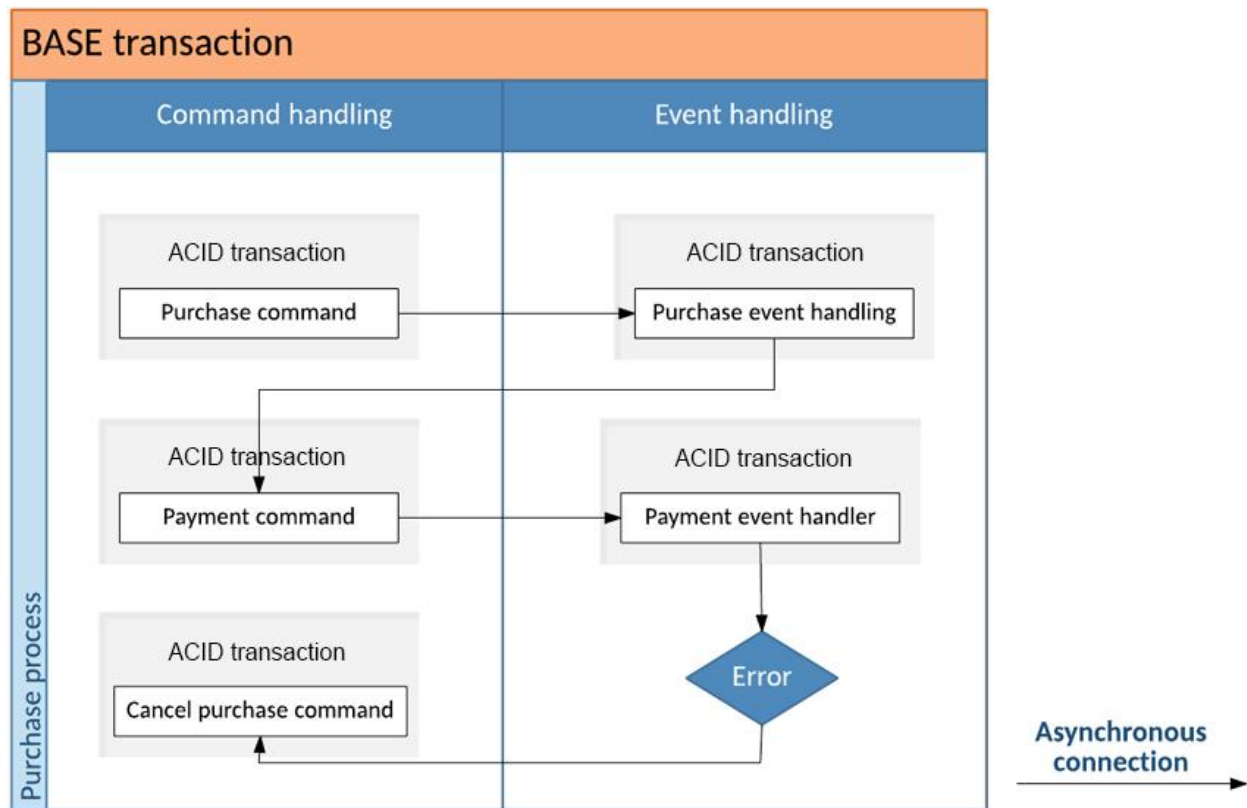
Basically available—The system must ensure the availability of data. There will be an answer for every request.

Soft state—The state of the system can change over time, even during periods where there is no input. That way, the system state is always “soft.” The consistency of the data needs to be managed by the developer and not by the database.

Eventually consistent—When new data is added to the system, it spreads gradually, one node at a time, to make the whole system consistent. The system eventually becomes fully consistent once it stops receiving input.

One result of this is the need to rethink the transaction rollback. In the BASE model, the transaction rollback is not automatic; you must design appropriate compensation actions for it. For example, if a user confirms his cart and buys the items, the application executes the related purchase command. If it is committed (as an ACID transaction), you can no longer act on it. Everything that happens next is asynchronous with respect to this command execution. Now, imagine that the listener of the purchase event has the purpose of starting the payment process. If the payment is not successful, the purchase command cannot be rolled back. Instead, the purchase must be canceled and the user notified. Figure 3 illustrates this business process.

Figure 3. Transaction rollback process

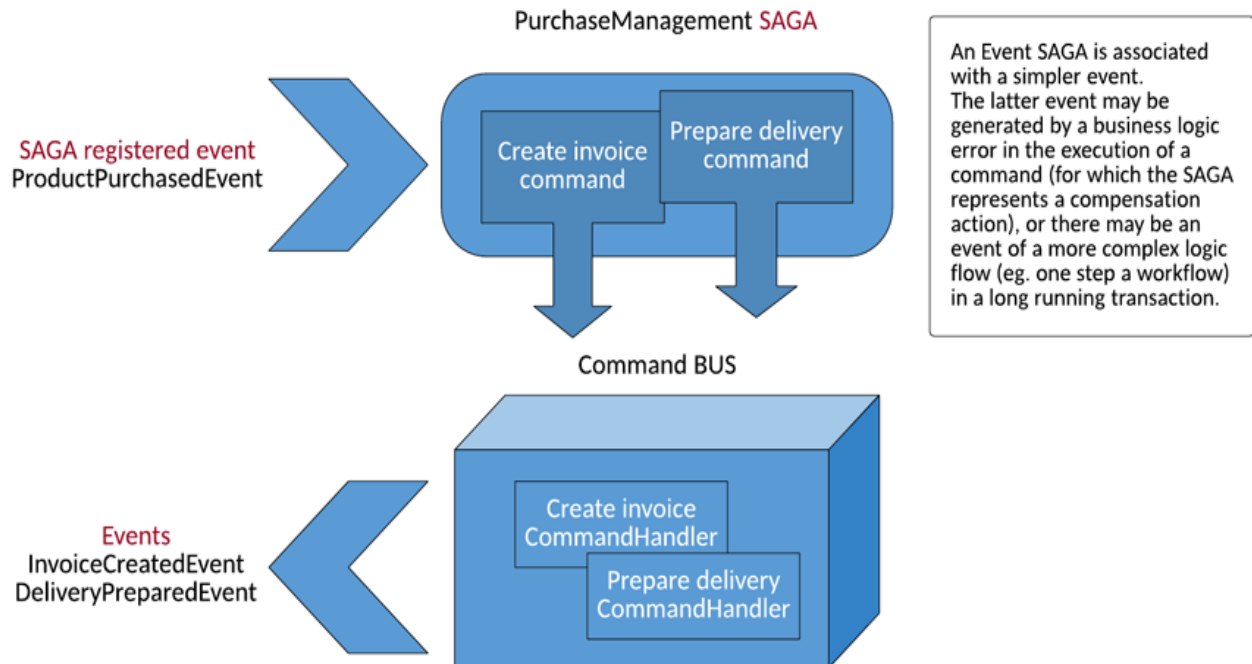


Step 5. Look for long-running transactions

Complex events, such as those discussed above, are managed by special components that are defined by the SAGA specification. SAGA is often defined as a “long-lived business transaction or process.” The foundational theory behind it is to avoid the use of blocking (and locking) transactions across lots of resources. In our purchase example, when a `ProductPurchased` event is generated in some way, a SAGA object associated with it sends a `CreateInvoice` command and a `PrepareDelivery` command. This is how the purchase process (or at least part of it) is executed.

Note that SAGA objects contain business behavior, but only in the form of process. This is a critical point: In their purest form, SAGA objects do not contain business logic.

Figure 4. SAGA example



In my example, I have written the following SAGA object to manage the order:

```
public class PurchaseManagementSaga extends AbstractAnnotatedSaga {

    @StartSaga
    @SagaEventHandler(associationProperty = "itemId")
    public void handle(ProductPurchasedEvent event) {
        // identifiers
        String deliveryId = createDeliveryId();
        String invoiceId = createInvoiceId();
        // associate the Saga with these values before sending the
commands
        associateWith("shipmentId", deliveryId);
        associateWith("invoiceId", invoiceId);
        // send the commands
        commandGateway.send(new PrepareDeliveryCommand(deliveryId));
        commandGateway.send(new CreateInvoiceCommand(invoiceId));
    }

    // ...
}
```

Show more

Step 6. Manage event sourcing

In addition to CQRS, the Axon Framework also implements an event sourcing pattern, to link commands to the queries. The use of event sourcing brings a lot of benefits, the most important of which is to have a system where everything is traced by design. (In a traditional application, you need to log to know who has done what.) All the events generated are persisted into the event store.

In my example, I have used a relational database for the event store, which is defined as follows.

```

<axon:jpa-event-store id="eventStore"
entity-manager-provider="myEntityManagerProvider"
event-serializer="eventSerializer" max-snapshots-archived="2"
batch-size="1000"/>

<bean id="myEntityManagerProvider"
class="org.axonframework.common.jpa.ContainerManagedEntityManagerProvider" />

```

Here is the event store's datasource definition.

```

<bean id="dataSource"
class="org.springframework.jndi.JndiObjectFactoryBean">
<property name="jndiName" value="jdbc/mydb"/>
</bean>

```

The `entityManagerFactory` (the bean that provides the database connections) is defined this way:

```

<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
<property name="persistenceUnitName" value="eventstoredb"/>

<property name="jpaVendorAdapter">
<bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
<property name="databasePlatform" value="${hibernate.sql.dialect}"/>
<property name="generateDdl" value="${hibernate.sql.generateddl}"/>
<property name="showSql" value="${hibernate.sql.show}"/>
</bean>
</property>
<property name="dataSource" ref="dataSource"/>
</bean>

```

Show more

In the event store, you can find the tables `DOMAINEVENTENTRY` and `SNAPSHOTEVENTENTRY` (created by the framework the first time), each of which has the following columns:

Figure 5. DOMAINEVENTENTRY and SNAPSHOTEVENTENTRY table details

Column Name	Owner	Type Schema	Type	Ordinal Position	Length	Scale	String Length	Not Null	Default	Identity?	Generated?
AGGREGATEIDENTIFIER	DOMAINEVENTENTRY	SYSIBM	VARCHAR	0	255	0	255 OCTETS	<input checked="" type="checkbox"/>		<input type="checkbox"/>	
SEQUENCENUMBER	DOMAINEVENTENTRY	SYSIBM	BIGINT	1	8	0		<input checked="" type="checkbox"/>		<input type="checkbox"/>	
TYPE	DOMAINEVENTENTRY	SYSIBM	VARCHAR	2	255	0	255 OCTETS	<input checked="" type="checkbox"/>		<input type="checkbox"/>	
EVENTIDENTIFIER	DOMAINEVENTENTRY	SYSIBM	VARCHAR	3	255	0	255 OCTETS	<input checked="" type="checkbox"/>		<input type="checkbox"/>	
PAYLOADREVISION	DOMAINEVENTENTRY	SYSIBM	VARCHAR	4	255	0	255 OCTETS	<input type="checkbox"/>		<input type="checkbox"/>	
PAYLOADTYPE	DOMAINEVENTENTRY	SYSIBM	VARCHAR	5	255	0	255 OCTETS	<input checked="" type="checkbox"/>		<input type="checkbox"/>	
TIMESTAMP	DOMAINEVENTENTRY	SYSIBM	VARCHAR	6	255	0	255 OCTETS	<input checked="" type="checkbox"/>		<input type="checkbox"/>	
METADATA	DOMAINEVENTENTRY	SYSIBM	BLOB	7	255	0		<input type="checkbox"/>		<input type="checkbox"/>	
PAYLOAD	DOMAINEVENTENTRY	SYSIBM	BLOB	8	255	0		<input checked="" type="checkbox"/>		<input type="checkbox"/>	

Some sample content is shown in Figure 6.

Figure 6. DOMAINEVENTENTRY table content sample

	SEQUENCENUMBER	TYPE	EVENTIDENTIFIER	PA	PAYLOADTYPE	TIMESTAMP	METADATA	PAYLOAD
1	12	0	ShoppingCart	269e27d5-8496-4e64-8825-3c3a3808e2b7	[NULL]	cqrs.example.shoppingcart.event.ItemAddedEvent	2016-05-05T18:26:48.093+02:00	[BLOB]
2	123	0	ShoppingCart	b187a04a-820a-409d-9f6d-e1994b1b268c	[NULL]	cqrs.example.shoppingcart.event.ItemAddedEvent	2016-05-05T18:25:16.233+02:00	[BLOB]
3	13	0	ShoppingCart	49127c2d-3afd-4f65-875d-0ee62451dd00	[NULL]	cqrs.example.shoppingcart.event.ItemAddedEvent	2016-05-05T18:30:13.273+02:00	[BLOB]

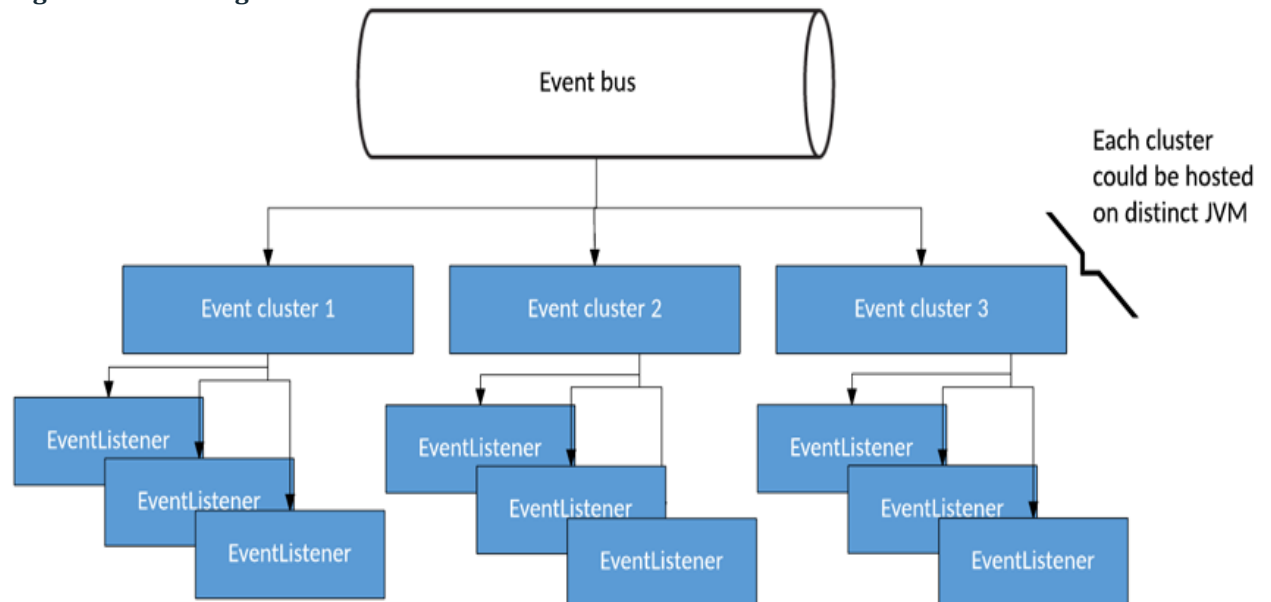
The event bus is the mechanism that dispatches the events to their listeners. Each event can have one or more listeners associated with it. Using the default implementation, you would write:

```
<axon:event-bus id="eventBus"/>
```

The framework makes two implementations available:

- The `SimpleEventBus` is applicable in most cases where dispatching is done synchronously and locally (such as in a single JVM).
- The `ClusteringEventBus` is suitable when your application requires `Events` to be published across multiple JVMs (that is, in a cloud ecosystem). The dispatching is therefore asynchronous and distributed. Figure 7 illustrates a clustering event bus.

Figure 7. Clustering event bus



Here is how that solution is declared.

```

<!-- Create an async cluster -->
<axon:cluster id="async">
  <!-- The inner bean defines the actual cluster implementation to use. Here,
  it is
  an asynchronous cluster. -->
  <bean class="org.axonframework.eventhandling.async.AsynchronousCluster">
    <constructor-arg value="async"/>
    <constructor-arg ref="asyncExecutor"/>
    <constructor-arg>
      <bean class="org.axonframework.eventhandling.async.FullConcurrencyPolicy"/>
    </constructor-arg>
  </bean>
  <!-- Here, we define which handlers we want to be part of this cluster. -->
  <axon:selectors>
    <axon:package prefix="cqrs.example.shoppingcart.eventhandler"/>
  </axon:selectors>

```

```

</axon:cluster>

<!-- We also create a simple cluster and we define it as default, meaning it
will
be selected when no other selectors (or clusters) match. -->
<axon:cluster id="simple" default="true"/>

<!-- We need a thread pool to execute tasks. -->
<bean id="asyncExecutor"
class="org.springframework.scheduling.concurrent.ThreadPoolExecutorFactoryBea
n">
    <property name="corePoolSize" value="1"/>
    <property name="waitForTasksToCompleteOnShutdown" value="true"/>
</bean>

```

Show more

Note the declaration of selectors. These allow you to select the cluster to associate the listeners to. (All the listeners in the package `cqrs.example.shoppingcart.eventhandler` are processed with the cluster named “async.” All the other listeners are processed with the cluster named “simple,” which is the default.)

`ClusteringEventBus` can define an `EventBusTerminal` as well. That way, you can distribute events to transmit events to an AMQP-compatible message broker, such as Rabbit MQ. I have used this capability in one of my application configurations (with only configuration and without code changes). See ” [IBM Cloud Container deployment](#)” for the deployment of this solution.

Step 7. Choose event serialization

It is possible to specify the type of serialization that can be in XML format, such as that shown in the example, or JSON format (using `org.axonframework.serializer.json.JacksonSerializer`), Java object serialization.

The serializer is used by the event store to serialize and deserialize event objects.

```




<bean id="eventSerializer"
class="org.axonframework.serializer.xml.XStreamSerializer"/>


```

Step 8. Interact with the application

To interact with a service, you can implement REST services, separating the command service (HTTP Put Request) from the query services (HTTP Get Request). I have created a front-end web application to activate the services, as shown in Figure 8.

Figure 8. Shopping cart front end

Choose Item	My Cart	Payment and Delivery	Confirm
<div>+</div>  Nephrolepis: price 10E			
<div>+</div>  monstera: price 10E			
<div>-</div>  white gardenia: price 10E			



Gardenia plants are prized for the strong sweet scent of their flowers, which can be very large in size in some species. Gardenia jasminoides (syn. G. grandiflora, G. Florida) is cultivated as a house plant. This species can be difficult to grow because it originated in warm humid tropical areas. It demands high humidity to thrive, and bright (not direct) light. It flourishes in acidic soils with good drainage and thrives on [68-74 F temperatures (20-23 C)][5] during the day and 60 F (15-16 C) in the evening. Potting soils developed especially for gardenias are available. G. jasminoides grows no larger than 18 inches in height and width when grown indoors. In climates where it can be grown outdoors, it can attain a height of 6 feet. If water touches the flowers, they will turn brown

availability: 10

Select quantity

1

price: 10E

add to cart

The user interface interacts through a servlet, in which the commands are dispatched by a `CommandGateway`:

```
.....
    private ApplicationContext ac;
    private CommandGateway commandGateway;

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
     response)
     */
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        response.setContentType("text/html");
        String itemId=request.getParameter("id");
        Integer quantity=Integer.parseInt(request.getParameter("quantity"));
        AddItemCommand addItemCommand=new AddItemCommand(itemId);
        addItemCommand.setQuantity(quantity);

        logger.debug("quantity:" + quantity);
        logger.debug("id:" + itemId);
        CommandCallback commandCallback = new CommandCallback<Object>() {
            @Override
            public void onSuccess(Object result) {
                logger.debug("Expected this command to fail");
            }
        }
    }
}
```

```

    }

    @Override
    public void onFailure(Throwable cause) {
        logger.debug("command exception", cause.getMessage());
    }
};
//asynchronous call - with command callback
commandGateway.send(addItemCommand, commandCallback);
}

@Override
public void init(ServletConfig config) throws ServletException {
    super.init(config);

    ac = new ClassPathXmlApplicationContext("/axon-db2-
configuration.xml");

    commandGateway= ac.getBean(CommandGateway.class);

}
.....
Show more

```

In the servlet initialization, I load the configuration of all Axon components, contained in a Spring configuration file.

In addition, you might note that the commands are dispatched through the command gateway. In the example, I send an `AddItemCommand` in an asynchronous way, without waiting for the process to end. I have registered a command callback to be notified of an eventual exception. This allows the system to notify the user if the real availability in stock is less than the quantity requested. (This might happen if another user adds the item to his own shopping cart while the first user is consulting the catalog.)

Step 9. The deployment

An application built with microservices allows you to choose different deployment solutions. You are often forced to deploy monolithic applications on dedicated machines. In the case of microservices, however, the deployment can be accomplished either on the same machine or distributed among several machines, depending on such requirements as performance, security, and scalability. Considering that the Axon Framework gives you the ability to change the configuration without changing any lines of code (other than the Spring file), along with the loose coupling of application components, you can see how many degrees of freedom you have for the application deployment. Each of the application components—command bus, event bus, event repository, command handling, event handling, and so on—can be hosted on a different combination of machines using different technologies. In this section, I show you a few of them.

IBM Cloud PaaS deployment

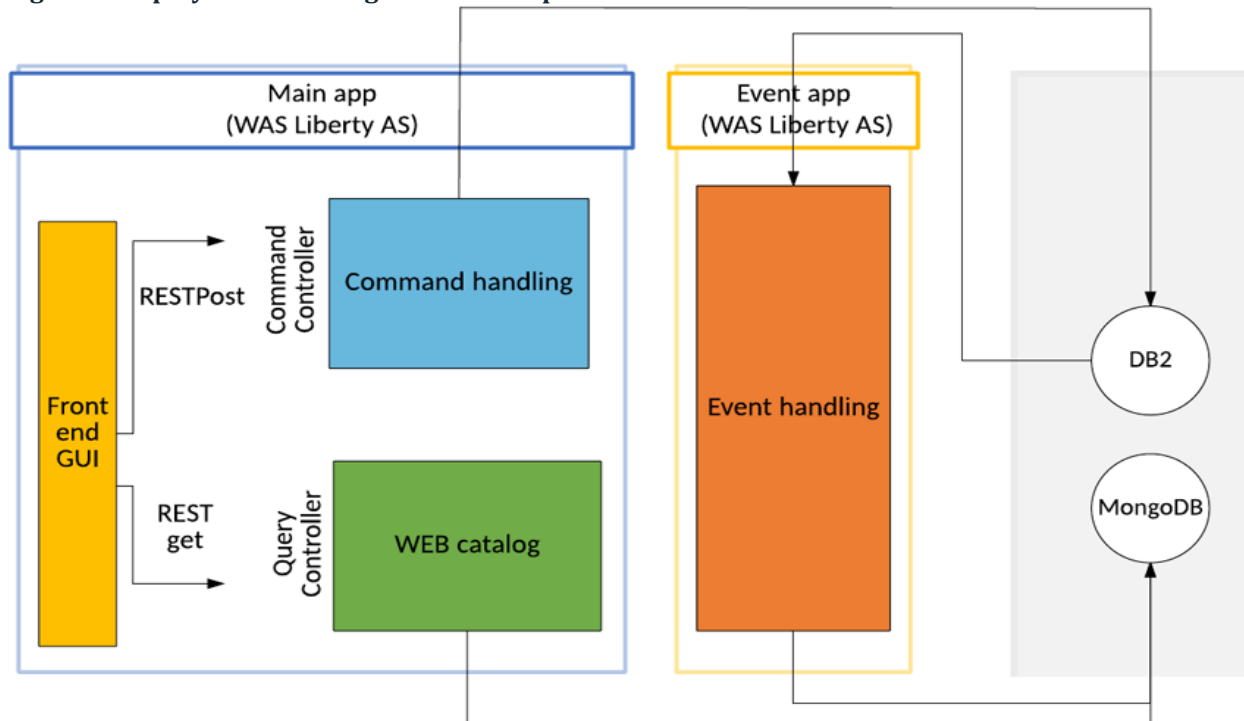
In deploying my application on IBM Cloud, the packaging consists of:

- A component for command handling—update requests (HTTP POSTs).
- A component for query handling—view requests, separate from command services (HTTP GETs).
- A component for event handling—contains the event listeners.

The application binds two IBM Cloud services (DB2 and MongoDB) for persistence of command data and event sourcing.

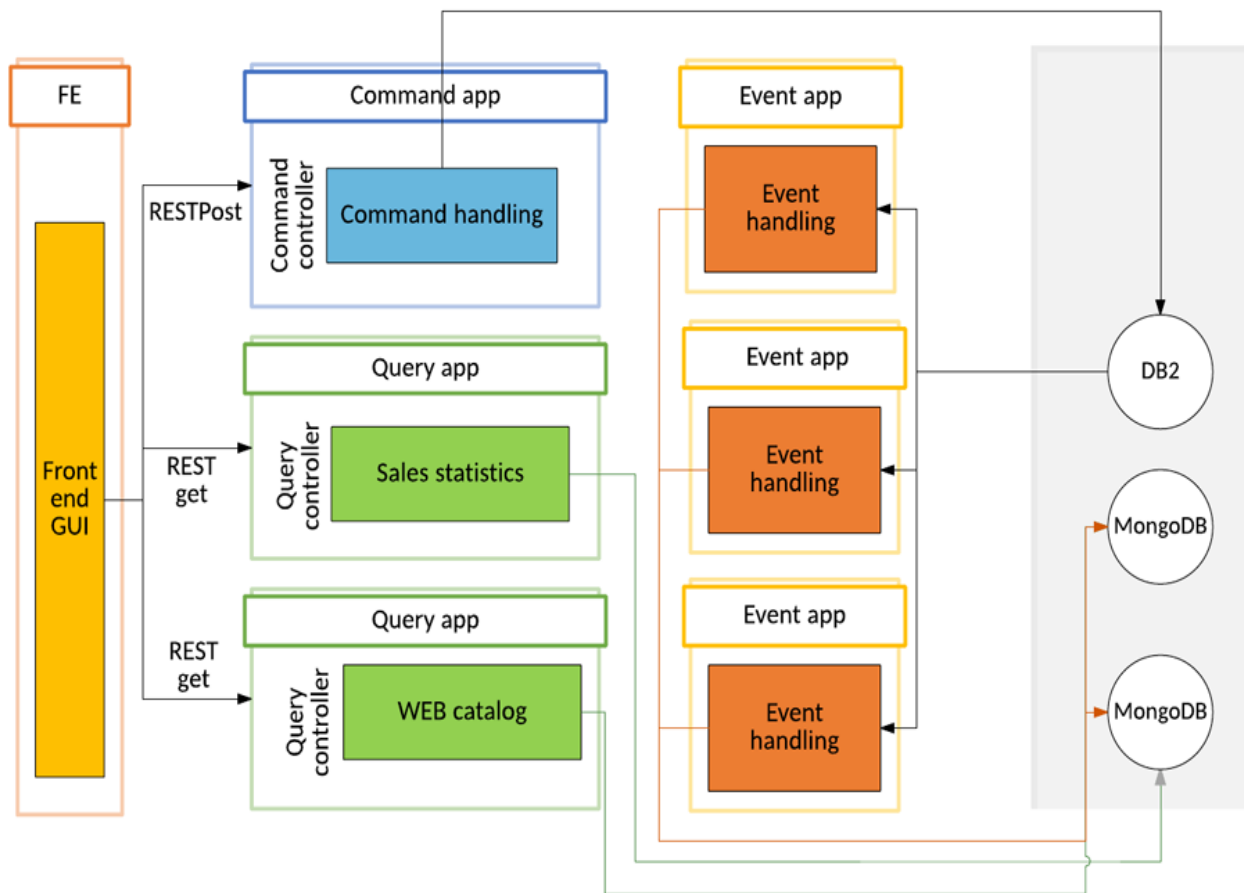
Figure 9 shows a possible deployment as a single application. This solution could be used for simplifying deployment and testing.

Figure 9. Deployment as a single macro component



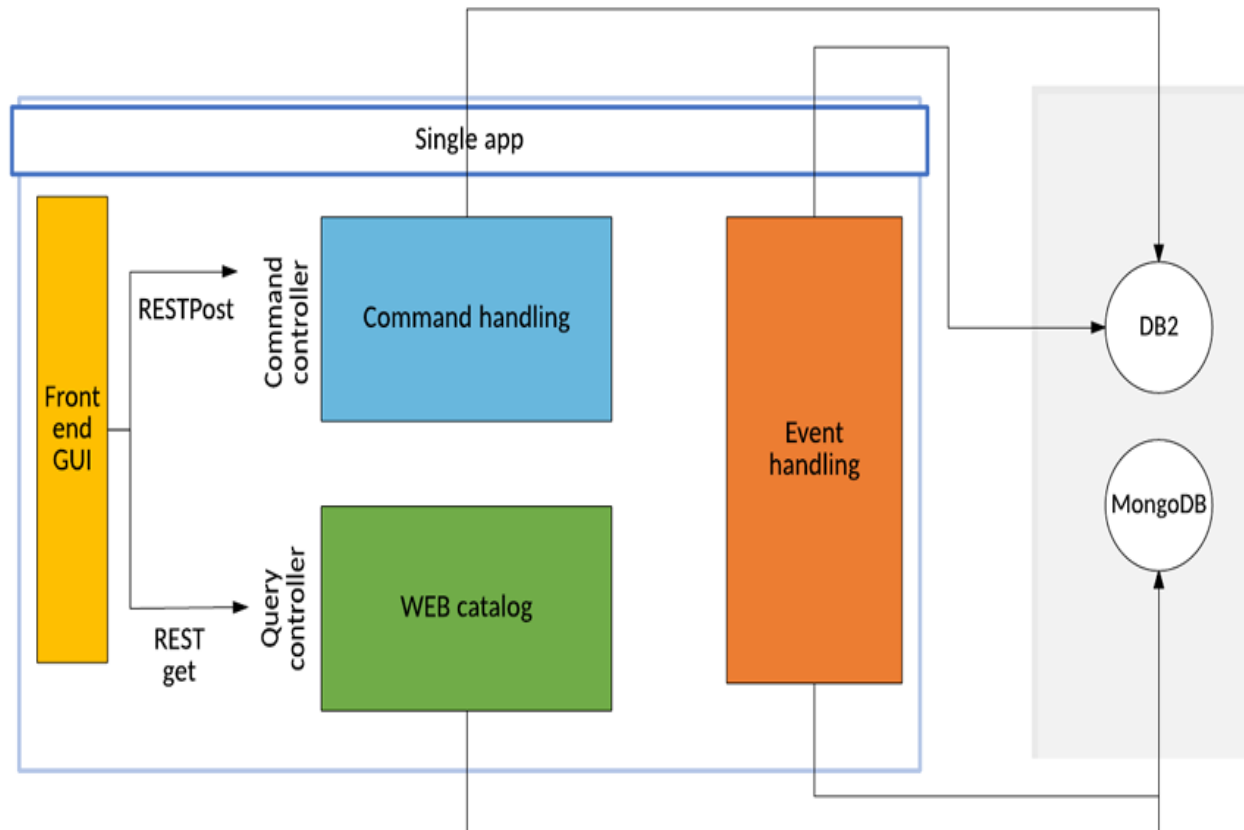
In Figure 10, I split the application into seven macro components, which allows me to provide the right resource level to each one. (For each of them, I could choose the number of VMs, the application language, the database service, and so on). Each of the macros can be deployed as a WAR file on an application server.

Figure 10. Deployment in macro components



Currently on IBM Cloud, I have deployed a halfway solution, using WebSphere Application Server Liberty for the application server, I have installed two web applications: one for the front end, command handling, and query handling, and another for event handling.

Figure 11. Deployment on IBM Cloud



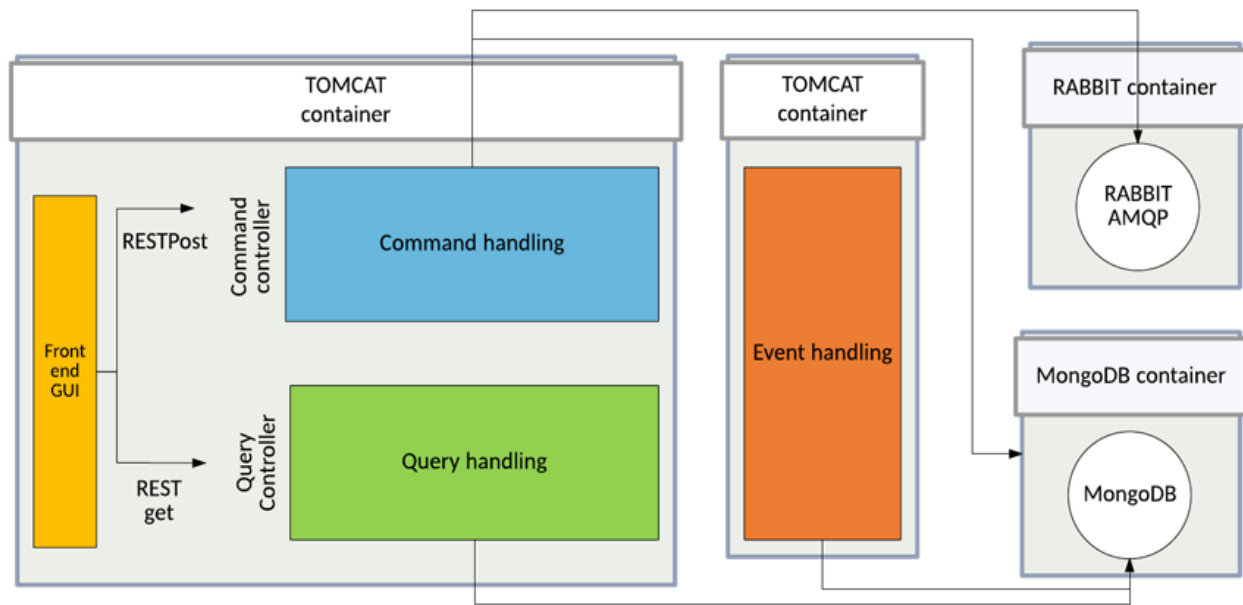
For this solution, I only have to package the application in a different way (I have two WAR files), but all the code is unchanged.

IBM Cloud container deployment

Another possibility is to deploy using IBM Cloud containers.

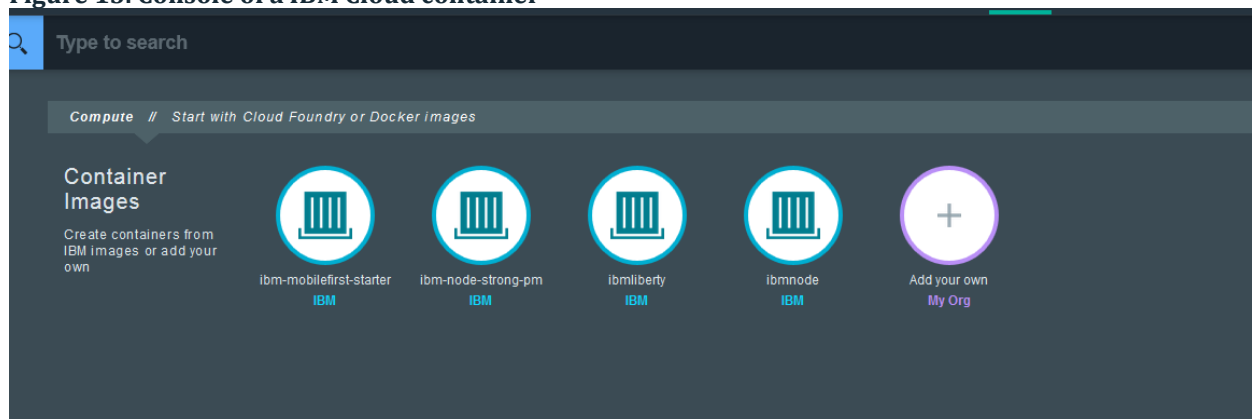
By changing only the Spring configuration file (and not the application code), you can change the deployment scheme. I believe this ability to adapt to the available environment is the most exciting characteristic of this pattern. In this case, I have used MongoDB for the event store and Rabbit-MQ for the event bus. Figure 12 shows how I organize the containers to host the application components.

Figure 12. Deployment with IBM Cloud containers



If this is your first time working with containers, log in to the IBM Cloud console and select **Add your own**.

Figure 13. Console of a IBM Cloud container



The first time you do this, you must set a name for the namespace that is associated with your private IBM Cloud repository. The namespace is used to generate a unique URL. Then you will receive all the instructions to access the IBM Cloud container space.

For a complete guide to managing the Docker container on IBM Cloud, see: "[IBM Containers plug-in \(cf ic\).](#)"

You need the Cloud Foundry CLI. To install the IBM Cloud Docker plugin (I installed it on my Ubuntu installation), enter the following:

```
./cf install-plugin
https://static-ice.ng.bluemix.net/ibm-containers-linux_x64
/cf install-plugin https://static-ice.ng.bluemix.net/ibm-containers-linux_x64
```

Log in to your IBM Cloud space and then to the IBM Cloud container space, and then enter:

```
./cf ic login
```

Now you can act on your private registry, adding the images that the application needs. In addition to the application's source code, the code I have provided contains Docker files for creating and configuring Docker images.

Conclusion

This article has described a way of designing highly scalable and available applications that are based on microservices built with polyglot persistence, event sourcing, and the CQRS pattern. By using the Axon Framework, you can focus on the business domain and write code that is ready for various configuration deployments, which hides the infrastructural complexity and makes the coding relatively easy. In fact, regardless of the complexity of the business logic, with these techniques the implementation of new commands, events, and event listeners can become essentially a purely mechanical and repetitive process.