WRITE ONCE.
SCALE ANYWHERE.

Migrating from JEE to SBA

Technical Guide

Uri Cohen | August 10, 2008

Abstract

This document details the migration process from a typical JEE tier-based application to a full blown Space-Based Architecture implementation, based on GigaSpaces XAP. It is the result of a project carried out by GigaSpaces in 2008, to determine the basis for comparison between GigaSpaces Space-Based Architecture and the standard JEE Tier-Based Architecture. The project was conducted by Grid Dynamics, an independent consulting and engineering company, hired by GigaSpaces for that purpose.

Table of Contents

Overview

The following document elaborates the steps required to migrate an OLTP enterprise application from a classic JEE Tier-Based Architecture (TBA), to a GigaSpaces Space-Based Architecture (SBA). It is the result of a project conducted by GigaSpaces, to determine a basis for comparison between GigaSpaces SBA and the standard JEE TBA. It also shows a gradual transition of a typical application from JEE to SBA. The results of this effort are described in full in a separate document (Please contact us directly for more details).

The transition from the baseline JEE application to full SBA, requires 3 main changes to the tier-based application, which will be described in detail below.

Motivation

This document is part of a project conducted in GigaSpaces to determine and quantify the overall benefits of migrating from JEE to SBA.

In general, we found that migrating the application from to JEE to SBA, can achieve a major performance boost, both in terms of throughput and latency. Additionally, as described below, it has significant advantages in terms of total cost of ownership, management overhead and ease of development.

Migration Effort

Generally speaking, a well architected enterprise application should have a business logic layer, a domain model layer and a data access layer. The runtime platform is responsible for providing ways for clients to invoke the business layer components, and managing other runtime aspects, such as transaction management, deployment, etc.

Here is a brief overview of the changes we applied to each of the above layers during the migration process:

1. Data Virtualization: Replace the database with a GigaSpaces IMDG. We modified the DAO implementation to access GigaSpaces instead of Hibernate.
2. Messaging Virtualization: Use the cache as a messaging tier with GigaSpaces JMS support on the feeder side, and polling event containers to listen to messages on the server side. Feeders use the GigaSpaces IMessageConverter support, to implicitly translate JMS messages to POJOs on the feeder side. and write the POJOs to the space, to be consumed by the polling containers.
3. Application & Deployment Virtualization: Collocate the space and the polling container, in the form of a GigaSpaces processing unit, thus implementing full SBA.

The following table shows a high level summary of the effort involved in each of the above steps, and the changes that are required:

Step	Config Change	Code Change	Effort (3 is biggest)
Data Virtualization	+	+*	3
Messaging Virtualization	+	-*	2
Application & Deployment Virtualization	+	-	1

* This depends on how the architectural layers of the application are isolated. If the data access code of the application is well isolated from other layers, typically it will be the only layer that needs to be changed. Similarly, If the JEE messaging components (typically EJB message driven beans) delegate business logic operations to POJOs, none or very minor code changes are required

1. Application Description

This section describes the application that is used throughout the document to demonstrate the transition from JEE to SBA.

1.1. General

The application is a simplified order management system. It has the following components:

1. Domain model layer: Contains the domain objects. In our case there are 3 such objects:
    a. **Symbol**, which represents a tradeable security
    b. **Order**, which represents an order made by a client of the system. There can be bid and ask orders.
    c. **Trade**, which represents a trade that occurred in the system, i.e. a bid and ask order that were matched.
2. Client side feeder: Simulates feeds from clients. Randomly generates bid and ask orders.
3. Business logic services: These services stand at the core of the system. They are invoked when new orders enter the system, and are in charge of validating the orders, and matching them to one another to create trades.
   The validator service performs basic validation on the incoming orders. Validation includes checking that the symbol is valid, the price is positive, etc.
   The matcher service matches bid orders with ask orders, and generates a trade object upon a successful match.
4. Data access layer: A set of interfaces to allow the business logic services to interact with the data store (relational database in the JEE case, IMDG in the case of SBA). These interfaces abstract the underlying data store implementation from the business logic services.
5. Messaging agents: These are classes that are responsible for interacting with the messaging infrastructure, and delegating the processing to them whenever a new event occurs in the system. (In our case this can be one of two – order submission by the feeder, or order validation success by the order validator service.) In the JEE case, the messaging agents are implemented as EJB3.0 Message Driven Beans that interact with the application server's JMS queue. In GigaSpaces no application code is required, as they are implemented in the form of OpenSpaces polling event containers.

1.2. Validation and Matching Logic

When an order is written into the system, it triggers the following phases:

- **Validation**: This is applied by the order validator. It ensures that the order's symbol is valid (i.e. it is contained in the symbol list) and that the quantity and price are not negative. After the validation is finished, bid orders are saved to the data store (DB or IMDG), and ask orders are returned back to the messaging system for the next phase (matching).
- **Only for ask orders**: Once an ask order is validated, it triggers the matching process for that order. In the case of the tier-based architecture, this is done by wrapping the validated ask order in a JMS message, and sending the message to the JMS queue. In the case of SBA, the validated ask order is simply written back to the IMDG, and triggers a matching phase by using the IMDG messaging capabilities.
- **Matching**: This is done by querying the data store (DB or IMDG) for a bid order with the same symbol, quantity and amount. This will always return an order due to the way the feeder works, and ensures deterministic behavior of the system across test cycles.
- **Trade creation**: Once the matching process is completed, the matched bid and ask orders are removed from the data store (DB or IMDG), and a trade object is written to it which contains the

details of the match (symbol, price, quantity, bid and ask order IDs).

- **Trade persistency**: In all architectures, trades eventually end up in the database and are never deleted from it.

1.3. JEE Implementation

The JEE implementation uses EJB and JMS, and is wired and configured through the Spring framework (Spring is very popular in such scenarios and is included in most enterprise project we've ran into). It includes the following layers:

- Domain model layer, which consists of POJOs, and implements the domain model for the application (Symbol, Order and Trade object types).
- Business services (Validator and Matcher), which are implemented as POJOs, and are activated by Message Driven Beans. The business services layer also includes a DAO interface, which enables operating on domain objects stored in the database. It isolates the business logic from the underlying database access code. The baseline implementation uses Hibernate to access the relational database.
- MDBs delegate the processing to the matcher and validator POJO services, which in turn use the DAO implementation to save the data to the database.
- A feeder application, which randomly creates orders with a random symbol, quantity and price. The orders are created in pairs, such that every bid order has a matching ask order and the matching process is deterministic.

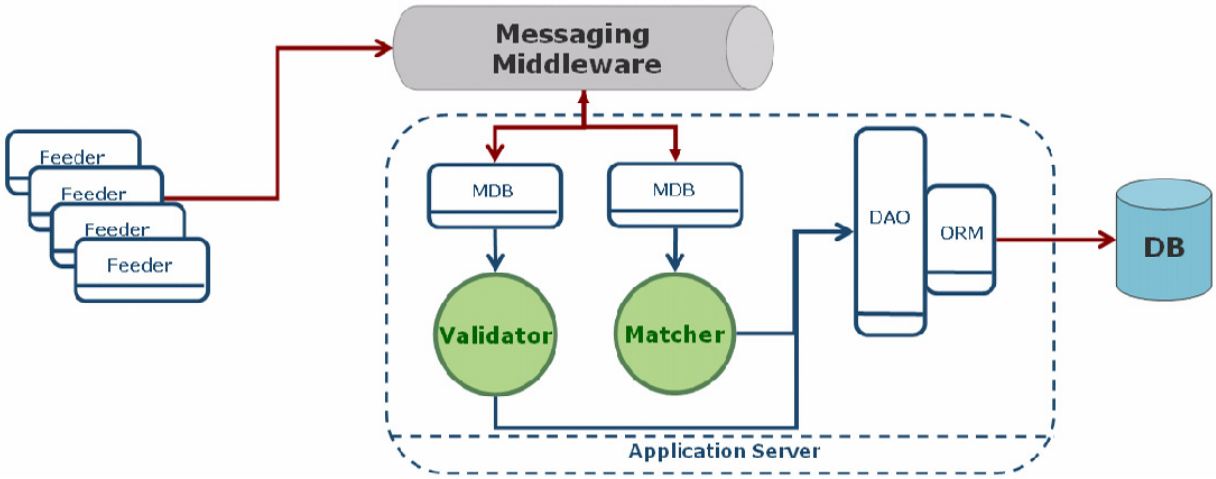

Figure 1 – JEE Implementation

Flow of execution of JEE implementation:

1. The feeder sends an order message to the appropriate queue.
2. The validator service receives the order via an EJB 3.0 message driven bean (MDB), and validates it. If the order is an ask order, it wraps it with a JMS message and sends the message to the queue. If it's a bid, it creates an order record in the DB using the DAO.
3. The matcher retrieves the validated ask order message from the queue via an MDB, and finds its matching bid order. It then deletes the bid order from the database ,and creates a new trade record in

it, using the DAO.

JEE implementation analysis:

The JEE implementation has the following flaws:

- There is an inherent network latency that cannot be avoided – it can be seen in the above diagram. All the arrows in dark red represent network hops, and it takes 5 network hops to process each order.
- Every network hop also incurs a serialization overhead when translating the data back and forth, so that it can be transferred over the network .
- The various tiers impose an overhead of translation between the different representation models, which are JMS message, Java Object and a relational database row. This translation is applied a few times for every transaction.
- The messaging server and the database are resources that are part of a single transaction, which requires the use of 2-phase-commit via the Java Transaction API to coordinate these two resources. This in turn imposes a significant overhead compared to a simple resource-local transaction.
- Both the messaging server and the database access the disk to maintain resiliency. This is very costly in terms of latency.

The above architecture also means that managing and deploying the application is not trivial, due to the many moving parts, specifically in clustered environments which aim to provide high availability.

1.4. SBA Implementation

The SBA implementation is deployed as a GigaSpaces processing unit with the Space collocated with the business logic. The Space is deployed in a partitioned-sync2backup topology. GigaSpaces event containers are used to deliver events to the business logic services. A mirror service is also deployed and is used to propagate completed trades to the database.

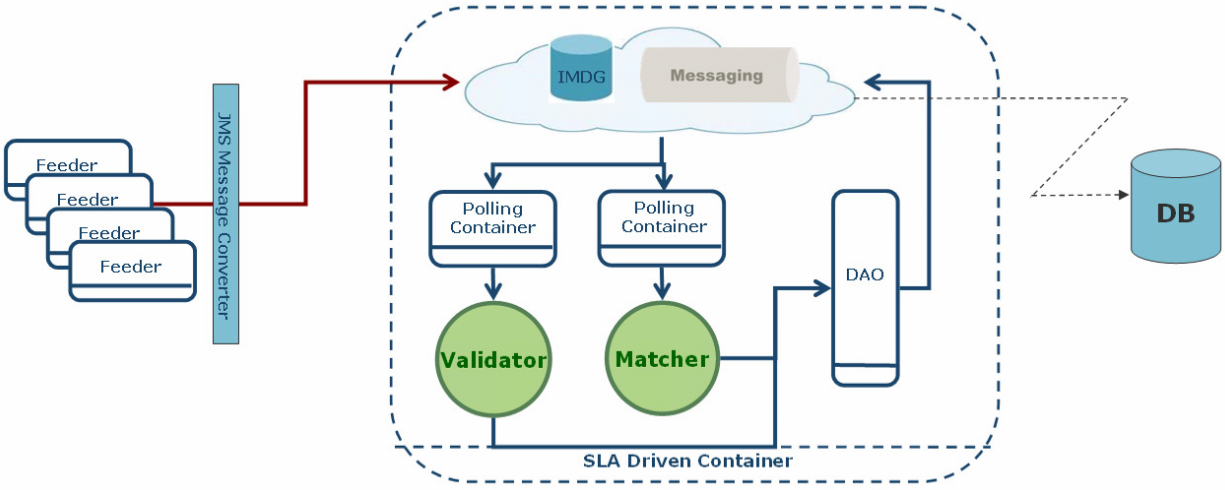

Figure 2 – SBA Implementation

SBA implementation flow of execution:

1. The feeder writes the order object to the IMDG. It uses the JMS API, but the GigaSpaces client side message converter (which is not exposed to the feeder code) converts the message to an order POJO and writes it to the space.

2. The validator service receives the order and validates it. It does so through the use of the GigaSpaces polling event container. It marks its state as validated, and writes it back to the IMDG. Note that this is done for both bid and ask orders.
3. The matcher retrieves the validated ask order message from the IMDG via the polling container, and finds its matching bid order. The IMDG is queried for the matching bid order. It then deletes the bid order from the IMDG, and creates a new trade record in it, using the DAO.
4. The trade record is propagated to the database asynchronously using the mirror service.

SBA implementation benefits:

- Network latency is reduced to a bare minimum, since the IMDG is now collocated with the business logic.
- Serialization overhead is reduced to a bare minimum, and only occurs when the feeder writes the order to the IMDG.
- Translation overhead remains at a bare minimum, and there are no format translations between the object oriented, relational and JMS model.
- 2PC is not needed; there is only one transactional resource (the space), which the transaction is using.
- Disk Utilization is brought to a bare minimum, and is only used asynchronously when persisting objects to the mirror.
- The number of moving parts is reduced significantly, since all components are deployed on the same platform. (There is no need for a separate JVM for the application and the IMDG, or for a separate messaging server.)

2. Virtualizing the Data Tier

This section describes how to convert the application's data tier from JEE to SBA, i.e. how to convert the data access code to work with GigaSpaces, instead of a relational database.

2.1. Data Access Interfaces

Our JEE baseline application is built with a data access layer (DAL). All other application components use the services of this layer to read data from, and write data to the database. As long as this separation of concerns is maintained, the DAL is the only application component that needs to be changed in this step.

There are two phases to this change:

The first is changing the DAL implementation itself to access the space instead of the database. The other is to wire the modified DAL implementation to the application components that are using it, namely the business logic services.

First, let's examine the data access interfaces that the application uses to store and save information:

```
public interface GenericDao<T> {
    List<T> list() throws DaoException;

    List<T> list(T template) throws DaoException;

    void save(T object) throws DaoException;

    void remove(T object) throws DaoException;
}
```

```
public interface OrderDao extends GenericDao<Order> {
    Order find(Symbol symbol, Boolean isBid, Double price) throws DaoException;

    Order find(Symbol symbol, Integer quantity, Boolean isBid,
               Double price, String status) throws DaoException;
}
```

```
public interface SymbolDao extends GenericDao<Symbol> {
    boolean isValid(Symbol symbol) throws DaoException;
}
```

```
public interface TradeDao extends GenericDao<Trade> {
}
```

In the above interface definitions, we see the 3 interfaces that our application is using to access and store data in the system (one interface for each object type used by the application). They all extend a generic data access interface that provides basic instance level operations, such as listing, saving and deleting an instance from the system.

Generally speaking, most DAO interfaces are not much more complicated than the one above. They typically include a few basic methods (defined above by the GenericDao interface), and a few other methods for more sophisticated querying operations.

2.2. Hibernate Implementation

Let's examine the Hibernate based implementation of the above interfaces.

First let's look at the GenericDao Hibernate implementation. Note that it is using Spring's HibernateDaoSupport to facilitate easier configuration, and use its out of the box Hibernate support. It is also declared abstract, since it has no meaning on its own, except when subclassed.

```
abstract class GenericDaoHibernate<T> extends HibernateDaoSupport
                                      implements GenericDao<T> {
    protected abstract T newTemplate();

    public List<T> list() throws DaoException {
        T template = newTemplate();
        return list(template);
    }

    public List<T> list(T template) throws DaoException {
        List list;
        try {
            list = getHibernateTemplate().findByExample(template);
        } catch (DataAccessException e) {
            throw new DaoException(e);
        }
        return list;
    }

    public void save(T object) throws DaoException {
        try {
            getHibernateTemplate().save(object);
        } catch (DataAccessException e) {
            throw new DaoException(e);
        }
    }

    public void remove(T object) throws DaoException {
        try {
            getHibernateTemplate().delete(object);
        } catch (DataAccessException e) {
            throw new DaoException(e);
        }
    }
}
```

The implementation is quite straightforward, delegating the actual work to Spring's HibernateTemplate. Note the abstract newTemplate() method, which is delegated to the subclasses, and is responsible for creating a template (or an Example object in the Hibernate lingo) that will match all the objects of a certain type. We then use Hibernate's example API to retrieve the matching objects.

Now let's examine the concrete implementations for this class. We begin with the OrderDao implementation:

```
public class OrderDaoHibernate extends GenericDaoHibernate<Order>
                                implements OrderDao {
    public Order find(Symbol symbol, Boolean isBid, Double price)
                                                    throws DaoException {
        try {
            List orders = getHibernateTemplate().findByCriteria(
                    DetachedCriteria.forClass(Order.class).add(
                            Expression.eq("symbol", symbol)).add(
                            Expression.eq("isBid", isBid)).add(
                            Expression.eq("price", price)), 0, 1);
            return (Order) ((orders.size() > 0) ? orders.get(0) : null);
        } catch (DataAccessException e) {
            throw new DaoException(e);
        }
    }

    public Order find(Symbol symbol, Integer quantity, Boolean isBid,
                      Double price, String status) throws DaoException {
        try {
            List orders = getHibernateTemplate().findByCriteria(
                    DetachedCriteria.forClass(Order.class).
                    add(Expression.eq("symbol", symbol)).
                    add(Expression.eq("quantity", quantity)).
                    add(Expression.eq("isBid", isBid)).
                    add(Expression.eq("price", price)).
                    add(Expression.eq("status", status)), 0, 1);
            return (Order) ((orders.size() > 0) ? orders.get(0) : null);
        } catch (DataAccessException e) {
            throw new DaoException(e);
        }
    }

    protected Order newTemplate() {
        return new Order();
    }
}
```

We can see that the Hibernate's Criteria API is used to retrieve objects matching the given parameters.

Here are the implementations of the SymbolDao and TradeDao. Note that in the SymbolDao, we simply use Hibernate's simple Session.get() method to retrieve a Symbol object, based on its symbol field:

```
public class SymbolDaoHibernate extends GenericDaoHibernate<Symbol>
                                implements SymbolDao {
    protected Symbol newTemplate() {
        return new Symbol();
    }

    public boolean isValid(Symbol symbol) throws DaoException {
        boolean b;
        try {
            Symbol s = (Symbol) getHibernateTemplate().get(Symbol.class,
                                                           symbol.getName());
            b = (s != null);
        } catch (DataAccessException e) {
            throw new DaoException(e);
        }
        return b;
    }
}
```

```
public class TradeDaoHibernate extends GenericDaoHibernate<Trade>
                               implements TradeDao {
    protected Trade newTemplate() {
        return new Trade();
    }
}
```

Next, let's see an example of how the application is actually using these interfaces to perform business logic operations. Below you can see the OrderMatcher business service code, which is responsible for matching the orders and creating the trade objects upon successful matching. You can see that it has references for both the TradeDao and the OrderDao for this purpose:

```
public class OrderMatcher {
    private OrderDao orderDao;
    private TradeDao tradeDao;

    public OrderMatcher() {}

    public OrderMatcher(OrderDao orderDao, TradeDao tradeDao) {
        this.orderDao = orderDao;
        this.tradeDao = tradeDao;
    }

    ...
```

And here's the code for the actual order processing:

```
public Trade processOrder(Order order) throws DaoException {
    Trade trade = null;
    if (!order.getIsBid()) {
        Order matchedOrder = matchOrder(order);
        if (matchedOrder != null) {
            trade = makeTrade(order, matchedOrder);
            tradeDao.save(trade);
            orderDao.remove(order);
            orderDao.remove(matchedOrder);
        }
    }
    return trade;
}

public Order matchOrder(Order order) throws DaoException {
    return orderDao.find(order.getSymbol(), order.getQuantity(),
                         !order.getIsBid(), order.getPrice(),
                         Order.STATUS_ACCEPTED);
}
```

Note that there's no data store dependent code in this service, making it extremely flexible and insensitive to runtime changes.

Lastly, let's take a look at how these objects are wired together through a Spring XML configuration file (the Hibernate session factory configuration is not included):

```
<bean id="tradeDao" class="com.gigaspaces.shapeshifter.dao.hibernate.TradeDaoHibernate">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>

<bean id="orderDao" class="com.gigaspaces.shapeshifter.dao.hibernate.OrderDaoHibernate">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>

<bean id="symbolDao" class="com.gigaspaces.shapeshifter.dao.hibernate.SymbolDaoHibernate">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>

<bean id="orderChecker" class="com.gigaspaces.shapeshifter.service.OrderChecker">
    <constructor-arg ref="symbolDao"/>
</bean>

<bean id="orderMatcher" class="com.gigaspaces.shapeshifter.service.OrderMatcher">
    <constructor-arg index="0" ref="orderDao"/>
    <constructor-arg index="1" ref="tradeDao"/>
</bean>
```

2.3. GigaSpaces Implementation

In order to replace the data access layer with GigaSpaces, all we need to do is implement the data access objects on top of GigaSpaces, and change the Spring configuration to wire the GigaSpaces implementation instead of the Hibernate one.

Let's have a look at the GigaSpaces DAO implementations first. We start with the GenericDao implementation:

```
abstract class GenericDaoGigaSpaces<T> implements GenericDao<T> {
    protected GigaSpace space;
    protected abstract T newTemplate();

    public void setSpace(GigaSpace space) {
        this.space = space;
    }

    public List<T> list() throws DaoException {
        return list(newTemplate());
    }

    public List<T> list(T template) throws DaoException {
        Object[] objs = space.readMultiple(template, Integer.MAX_VALUE);
        List<T> list = new ArrayList<T>(objs.length);
        for (Object obj : objs) {
            list.add((T) obj);
        }
        return list;
    }

    public void save(T obj) throws DaoException {
        space.write(obj);
    }

    public void remove(T object) throws DaoException {
        space.take(object);
    }
}
```

We see that the whole operation maps nicely to the GigaSpace interface. The only thing that looks a bit redundant is the conversion to java.util.List in the list(T) method. This is forced because the GigaSpace.readMultiple() method returns an array of objects, which can only be converted to a generic List manually.

Next, let's review the OrderDao GigaSpaces implementation:

```
public class OrderDaoGigaSpaces extends GenericDaoGigaSpaces<Order>
                                implements OrderDao {
    public Order find(Symbol symbol, Boolean isBid, Double price) {
        Order templ = new Order();
        templ.setSymbol(symbol);
        templ.setIsBid(isBid);
        templ.setPrice(price);
        return space.read(templ);
    }

    public Order find(Symbol symbol, Integer quantity, Boolean isBid,
                      Double price, String status) throws DaoException {
        Order templ = new Order();
        templ.setSymbol(symbol);
        templ.setIsBid(isBid);
        templ.setPrice(price);
        templ.setQuantity(quantity);
        templ.setStatus(status);
        return space.read(templ);
    }

    protected Order newTemplate() {
        return new Order();
    }
}
```

As you can see, the implementation is quite straightforward. Both find() methods use the GigaSpace template matching mechanism, but could just as easily be implemented using GigaSpaces SQL query support. The implementation of the other two DAO interfaces is quite similar, and is shown below:

```
public class SymbolDaoGigaSpaces extends GenericDaoGigaSpaces<Symbol> implements SymbolDao {
    protected Symbol newTemplate() {
        return new Symbol();
    }

    public boolean isValid(Symbol symbol) {
        Symbol s = space.read(symbol);
        return (s != null);
    }
}
```

```
public class TradeDaoGigaSpaces extends GenericDaoGigaSpaces<Trade> implements TradeDao {
    protected Trade newTemplate() {
        return new Trade();
    }
}
```

Finally, let's review the Spring configuration for wiring the GigaSpaces DAO implementation with the business logic services:

```
<os-core:space id="dataSpace" url="/./shape-shifter" ...>
    ....
</os-core:space>

<os-core:giga-space id="gigaSpace" space="dataSpace" tx-manager="transactionManager"/>

<bean id="tradeDao" class="com.gigaspaces.shapeshifter.dao.gigaspaces.TradeDaoGigaSpaces">
    <property name="space" ref="gigaSpace"/>
</bean>

<bean id="orderDao" class="com.gigaspaces.shapeshifter.dao.gigaspaces.OrderDaoGigaSpaces">
    <property name="space" ref="gigaSpace"/>
</bean>

<bean id="symbolDao" class="com.gigaspaces.shapeshifter.dao.gigaspaces.SymbolDaoGigaSpaces">
    <property name="space" ref="gigaSpace"/>
</bean>

<bean id="orderChecker" class="com.gigaspaces.shapeshifter.service.OrderChecker">
    <constructor-arg ref="symbolDao"/>
</bean>

<bean id="orderMatcher" class="com.gigaspaces.shapeshifter.service.OrderMatcher">
    <constructor-arg index="0" ref="orderDao"/>
    <constructor-arg index="1" ref="tradeDao"/>
</bean>
```

We can see that instead of the Hibernate implementation for the DAOs, we now use the GigaSpaces ones we saw above. This wraps up the required changes for changing the data access layer from Hibernate to GigaSpaces.

2.4. Other Considerations

Transaction Management

You might have noticed that we didn't mention how transactions are managed, and this was actually intentional. Since our application "lives" in a Spring environment, we let Spring manage our transactions. Both GigaSpaces and Spring's Hibernate template are aware of Spring transactions. (In fact, as of Hibernate 3.0.1 you can also use plain Hibernate interfaces, instead of HibernateTemplate, and also enjoy Spring declarative transaction support.) Therefore, any transaction that was initiated by Spring, or that Spring has identified, is also propagated to our data access objects, when called.

In our case, the business logic is invoked by the runtime platform's messaging agents (MDBs in the case of JEE, OpenSpaces event containers in the case of SBA). Both start a transaction when processing messages / objects that are written to the system, and this transaction is propagated to our data access objects, which do all of the operation in its transactional context.

If you choose to invoke your data access directly, you should initiate a Spring transaction (declaratively or explicitly) to make sure the code executes within a transactional. The key point here, is that both implementations support this model, and there's no need to change either the configuration or the code.

Complex Queries

In many cases, it is not simple to map Hibernate, JPA or even plain SQL99 queries to GigaSpaces queries, mainly because these query languages are richer and more expressive than GigaSpaces query support. In such cases, the preferred option is to use GigaSpaces dynamic language support, and perform the complex query as a number of queries expressed in a script, processed and executed on the server (similar to a stored procedure or database script sent from the client). You can use this scripting language to manipulate and extract the data you need from the space. In future versions of GigaSpaces, we plan to enhance our query support to allow for more complex queries.

Model Discrepancies

Hibernate has the notion of relationship between entities and fetching strategies, or referenced entities, i.e. object references can map to different physical tables.

With GigaSpaces, object references are saved by default in their serialized form, as part of the same physical tuple in the space. Therefore, by default, GigaSpaces does not deal with fetching strategies, and the entire object graph is retrieved with every read operation. For cases in which the space is embedded, this is not a significant drawback as everything is passed by reference anyway, and there is little sense in lazy loading a referenced object. For remote clients that rely on lazy loading of referenced objects for performance, this may become an issue.

The way to currently deal with this limitation, is to change the object model and hold the GigaSpaces entry IDs, instead of physical Java references. One can also imitate the mechanism that is used by Hibernate, and use smart stubs that load the actual object upon first usage by the client code. In general, we do not advise people to do that, since it may break the locality of the object graph, in such a way that referenced objects may belong to a different partition than their referencing objects. This creates performance overhead, and limits the latency and scalability of the system.

3. Virtualizing the Messaging Tier

In this section, we show how to convert the messaging tier of the application to use GigaSpaces. Typically, this can bring an increase of up to 6 times in throughput, and a decrease of 10 times in latency compared to traditional messaging systems that implement resiliency using disk persistency.

In a JEE environment, the JMS API is used to send and consume messages in the system.

A message sender typically creates a JMS message of a certain type (ObjectMessage, TextMessage, etc.) and sends it to a message destination, which is either a JMS queue or a JMS topic. At the other end, we have a message consumer, that receives the message by subscribing to the relevant destination, as providing a message selector if necessary to filter the consumed messages. In a JEE environment, this is typically a message driven bean component. With GigaSpaces, the space itself, besides serving as a data store, is also used as a messaging infrastructure. Sending a message is merely writing an object to the space. Consuming it can be done by either subscribing for notifications on it (based on a certain criteria for example), or retrieving it directly from the space via the take operations. Both are abstracted away from the client code, using OpenSpaces event containers, so all the client code has to implement, is the actual business logic for processing the event.

3.1. Server Side Messaging

We'll start by reviewing the server side JEE implementation of the messaging tier. As you may recall from the previous section, the business logic in our application is handled by the business logic services (OrderChecker and OrderMatcher). In the JEE implementation, these messages are invoked by message driven beans, which are managed by the EJB container in the application server environment. Let's see how our MDBs are configured for the JBoss application server (it is assumed that the matching JMS destinations and other resources are already configured by the application server). We'll show only the OrderMatcherMDB, as the OrderCheckerMDB is quite similar in content.

First, let's see how the MDB is configured using annotations:

```
@MessageDriven(
        activationConfig =
        {
            @ActivationConfigProperty
                (propertyName = "destinationType",
                 propertyValue = "javax.jms.Queue"),
            @ActivationConfigProperty
                (propertyName = "destination",
                 propertyValue = "queue/tba.MatcherQueue")
        }
)
@TransactionManagement(value = TransactionManagementType.CONTAINER)
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class OrderMatcherMdb implements MessageListener {

    @Resource(mappedName = "XAConnectionFactory")
    private ConnectionFactory connectionFactory;
    private Session outSession;

    @Resource(mappedName = "queue/tba.MatcherQueue")
    private Queue outNoMatchDest;
    private MessageProducer outNoMatch;
    private OrderMatcher orderMatcher;
```

In the above snippet we see that the MDB is configured to work with a JMS queue named "queue/tba.MatcherQueue". It is injected with the JMS connection factory and the queue instance itself, so that it can later send messages to it. Also, we choose to use container managed transactions, and use the REQUIRED transaction attribute. Naturally, we can also use XML to configure all of this. Next, let's see how the Spring application context is loaded, and a reference to the OrderMatcher business logic service is obtained:

```
@PostConstruct
public void init() {
    try {
        Connection connection = connectionFactory.createConnection();
        this.outSession = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        this.outNoMatch = this.outSession.createProducer(this.outNoMatchDest);
        // use spring to set orderMatcher
        initSpringBeans();
    } catch (JMSException e) {
        logger.fatal("failed to init", e);
        throw new RuntimeException(e);
    }
}

private void initSpringBeans() {
    BeanFactoryLocator locator = ContextSingletonBeanFactoryLocator.getInstance();
    BeanFactoryReference bfr = locator.useBeanFactory("tbaApplicationContext");
    BeanFactory factory = bfr.getFactory();
    orderMatcher = (OrderMatcher) factory.getBean("orderMatcher");
}
```

You can see that we use the @PostConstuct annotation to instruct the EJB container to call the init method once the MDB instance is created and injected with all the resources.
In this method we create a message producer which will later use, and initialize the Spring application context (this is the same application context XML file presented in the previous section). We then retrieve the OrderMatcher instance from it to do the actual matching. Next, let's see how the actual message handling is done:

```
public void onMessage(Message msg) {
    try {
        Order order = (Order)((ObjectMessage)msg).getObject();
        Trade trade = orderMatcher.processOrder(order);
        if (trade == null && !order.getIsBid()) {
            //no match for ask order - resend it to end of queue
            this.outNoMatch.send(this.outSession.createObjectMessage(order));
        }
    } catch (JMSException t) {
        ...
    } catch (DaoException e) {
        ...
    }
}
```

As you can see, this is quite straightforward, and most of the work is delegated to the OrderMatcher instance. This keeps our MDB simple and allows us later to replace it with GigaSpaces messaging infrastructure without changing any code. Note that if a match is not found, we send the order back to the queue to be matched later on.

So essentially, all that needs to be done, is to replace this MDB with a GigaSpaces specific messaging agent that would call the OrderMatcher when a message is received. We do that by using a GigaSpaces polling event container that will trigger the OrderMatcher when an order is written into the system.

We show how this can be done by directly annotating the OrderMatcher class. Note however, that this can also be done purely via XML, in case you don't feel like recompiling your code, or you are using Java 1.4.

```
@EventDriven
@Polling
@TransactionalEvent
public class OrderMatcher {
    private OrderDao orderDao;
    private TradeDao tradeDao;

    public OrderMatcher() {}

    public OrderMatcher(OrderDao orderDao, TradeDao tradeDao) {
        this.orderDao = orderDao;
        this.tradeDao = tradeDao;
    }

    @EventTemplate
    public Data unmatcherAskOrder() {
        Order template = new Order();
        template.setStatus(Order.STATUS_ACCEPTED);
        template.setIsBid(false);
        return template;
    }
```

Note the class level annotations, which classify this class as event driven, using a polling event container, with transactional support. These annotations will later be processed when this class is instantiated by the Spring application context. Also note the @EventTemplate annotation, which defines the template by which this event container filters events. In our case we only deal with ask order objects that are already validated (i.e. their status is accepted). Next, we need to instruct the event container how to process the event, in this case, the order. For that we annotate the processOrder method, which tells the event container to call it, once the event is received:

```
@SpaceDataEvent
public Trade processOrder(Order order) throws DaoException {
    Trade trade = null;
    if (!order.getIsBid()) {
        Order matchedOrder = matchOrder(order);
        if (matchedOrder != null) {
            trade = makeTrade(order, matchedOrder);
            tradeDao.save(trade);
            orderDao.remove(order);
            orderDao.remove(matchedOrder);
        }
    }
    return trade;
}
```

Last, let's view the Spring configuration for instantiating OrderMatcher and creating the event container based on these annotations:

```
<context:component-scan base-package="com.gigaspaces.shapeshifter.service"/>

<os-events:annotation-support />

<os-core:space id="space" url="/./space"/>

<os-core:local-tx-manager id="transactionManager" space="space"/>

<os-core:giga-space id="gigaSpace" space="space" tx-manager="transactionManager"/>
```

That's it. Spring scans through our classes on startup and initializes the right components. Note that if we still want to use dependency injection for our OrderMatcher (to inject the DAO implementation references for example) we either define it explicitly in the Spring XML file, or use annotations such as Spring's @AutoWired.

3.2. Client Side Messaging

On the client side, which is actually our feeder, we simply write JMS messages to the validation queue:

```
public class Feeder {
    JmsTemplate jmsTemplate;

    public void setJmsTemplate(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }

    public void run() {
        ...
        Order bid = new Order(...);
        jmsTemplate.convertAndSend(bid);
        Order ask = new Order(...);
        jmsTemplate.convertAndSend(ask);
        ...
    }
}
```

Messages are written using Spring's JmsTemplate, which is injected to the Feeder instance by Spring. In the JEE case, the configuration uses Spring's JNDI support:

```
<jee:jndi-lookup id="jmsQueueConnectionFactory" jndi-name="ConnectionFactory"/>
<jee:jndi-lookup id="sendDestination" jndi-name="${feeder.checker.queue}"/>

<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="jmsQueueConnectionFactory"/>
    <property name="defaultDestination" ref="sendDestination"/>
    <property name="receiveTimeout" value="30000"/>
    <property name="deliveryPersistent" value="true"/>
</bean>

<bean id="feeder" class="com.gigaspaces.shapeshifter.service.Feeder">
    ...
    <property name="numThreads" value="${feeder.threads}"/>
    <property name="jmsTemplate" ref="jmsTemplate"/>
</bean>
```

To have the feeder write to the GigaSpaces cluster instead of the JEE JMS queue, we use GigaSpaces JMS support. The idea behind it is to allow the feeder code to still use JMS, but under the hood, still on the feeder side, this message is transformed into a POJO and sent to the GigaSpaces cluster like any other POJO. The translation is done by an implementation of the interface IMessageConverter, to which the conversion work is delegated. In our case we use the simple ObjectMessage2ObjectConverter which assumes you're using a JMS ObjectMessage, and simply extracts the object from it, using the getObject method. Here's how this is defined:

```
<bean id="messageConverter" class="com.j_spaces.jms.utils.ObjectMessage2ObjectConverter"/>
<os-jms:connection-factory id="connectionFactory" giga-space="gigaSpace"
                           message-converter="messageConverter"/>
<os-jms:queue id="checkerQueue" name="tba.CheckerQueue"/>

<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="defaultDestination" ref="checkerQueue"/>
    <property name="receiveTimeout" value="30000"/>
</bean>

<bean id="feeder" class="com.gigaspaces.shapeshifter.service.Feeder">
    ...
    <property name="numThreads" value="${feeder.threads}"/>
    <property name="jmsTemplate" ref="jmsTemplate"/>
</bean>
```

To conclude this section; the above examples showed how you can migrate from a JMS implementation to a GigaSpaces one, such that no code needs to be changed.

3.3. Other Considerations

Full JMS API Support

Besides the client side JMS message conversion, supported by GigaSpaces, the product also supports most of the JMS 1.1 API. This page lists the current unsupported features in GigaSpaces JMS implementation.

4. Application & Deployment Virtualization

This section deals with how to take the modified code, use the GigaSpaces deployment infrastructure to simplify the deployment process, and utilize the SBA model in full.

Once the application has been changed to use GigaSpaces for both data access and messaging, the last step is to deploy it in the form of a GigaSpaces processing unit instead of the classic JEE deployment unit (typically an EJB JAR or .ear file).

To achieve true SBA we have to follow the guidelines below:

- All tiers need to be collocated. This means that the business logic services, and the space with which they interact, are part of the same processing unit.
- Ideally, the data needs to be partitioned in such a way that it enables transactions to be processed within the boundaries of a single JVM. This means that orders and symbol objects that represent the same stock end up in the same partition, and the business logic services described earlier, are available on every JVM.
- In addition, we need to make sure that the application is fault tolerant and has no single point of failure

4.1. Deployment Structure

A GigaSpaces processing unit (PU) is in essence, a Spring application that is bundled as a JAR file (much like an OSGi bundle). It has a predefined directory structure, and besides the application's compiled classes, it also contains the Spring configuration file (under META-INF/spring/pu.xml) and any 3rd party JAR that the application depends on. It can run from within your IDE (like any other Spring application), but more interestingly, it can run on GigaSpaces SLA driven containers. When deployed on the GigaSpaces SLA driven grid, the GigaSpaces deployment manager provisions the application according to predefined SLA definitions. These SLA definitions are part of the Spring configuration file, and are processed by the GigaSpaces deployment manager. Based on these definitions, the deployment manager decides how many instances of the processing unit are provisioned, and which physical containers to provision them to.

Let's see these SLA definitions, which are part of the SBA implementation pu.xml Spring configuration file:

```
<os-sla:sla cluster-schema="partitioned-sync2backup"
            number-of-instances="2"
            number-of-backups="1"
            max-instances-per-machine="1"/>
```

The above snippet controls the clustering topology of the space embedded in the processing unit, and the way the processing unit instances are provisioned. Here we chose to use the partitioned-sync2backup topology, with 2 primary PUs and 1 backup PU for each. This also means that the space embedded within this processing unit has two primary partitions and a backup partition for each on them.

The last element here is max-instances-per-machine. This means that no primary and its backup are deployed on the same physical host, thus ensuring that there's no single point of failure. The GigaSpaces deployment manager makes sure that the application is provisioned, based on these rules and deploys them on actual containers that are running in the network.

4.2. Controlling Object Routing

Let's examine the way we control how bid and ask orders for the same stock, end up in the same physical location. This is done by using the GigaSpaces support for object routing. The user determines for each class written to the space, the field which is used to control the routing. This means that two

instances of the same class are routed to the same physical location, provided that they have the same value for the field used for the routing. The designation of the routing field can be done either via annotations or XML. Here's how this is done for the Order object in our application (note the @SpaceRouting annotation):

```
@SpaceClass(persist = false)
public class Order implements Serializable {
    public Order() {}
    ....

    @SpaceId
    public Long getId() {
        return id;
    }

    @SpaceRouting
    public Symbol getSymbol() {
        return symbol;
    }
    ....
}
```

5. Future Directions

In this section, we discuss future directions that the GigaSpaces product can take, in order to further ease the migration process, and make it as painless and seamless as possible.

5.1. Web Container Support

The sample application we have used in this document (and in the migration project in general), did not contain a web tier. GigaSpaces 6.5 does not contain a way to deploy and use JEE web applications. However, it does contain features, such as remoting and JMS support that would enable a very smooth migration from a classic JEE web application that uses these mechanisms to invoke business services, to the SBA solution.

As of GigaSpaces 6.6, web container support will be built into the product. This means that JEE web applications can be deployed directly into the GigaSpaces runtime, and enjoy the SLA driven deployment capabilties of the GigaSpaces product. In addition, there will also be tighter integration between web applications that use GigaSpaces, and the GigaSpaces data grid itself, e.g. built in HTTP session replication, accessing processing unit constructs from within a standard web application, and more.

5.2. JPA and EJB3.0 Support

Another option for providing an even more seamless way of transitioning from JEE to SBA, is providing native GigaSpaces support for JPA and EJB3.0. Specifically, being able to plug in GigaSpaces under the JPA API, and being able to support the EJB3.0 contract for stateless session beans at the minimum.

This is a future direction that we are considering implementing, as part of the upcoming product releases, but it is not yet an official part of the product roadmap.

Summary

In this document we have shown how to migrate from JEE to SBA, with as few code changes as possible.
We have shown that when architected correctly, JEE applications lend themselves very well to the migration process, and can utilize the GigaSpaces XAP runtime environment to achieve significant improvements in both performance and scalability.

It should be noted that for applications that are not architected "by the book" and are not based on the Spring framework, the migration process might be more time consuming and involve more changes to the code.

GigaSpaces is working on further improving the XAP platform to make this transition as seamless and easy as possible.

U.S. Headquarters
GigaSpaces Technologies Inc.
317 Madison Ave, Suite 1220
New York, NY 10017
Tel: 646-421-2830
Fax: 646-421-2859

U.S. West Coast Office
GigaSpaces Technologies Inc.
555 California Street, 3rd Floor
San Francisco, CA 94104
Tel: 415-568-2125
Fax: 415-651-8801

Europe Office
GigaSpaces Technologies Ltd.
2 Sheraton Street,
London, W1F 8BH, UK
Tel: +44-207-117-0213
Fax: +44-870-3835-135

International Office
GigaSpaces Technologies Ltd.
4 Maskit Street, P.O. Box 4063
Herzliya, 46140, Israel
Tel: +972-9-952-6751
Fax: +972-9-957-6780