Transparently Clustered Spring—
A Runtime Solution for Java™ Technology

Jonas Bonér

Senior Software Engineer
Terracotta, Inc.

http://www.terracottatech.com

TS-3217

What You Will Learn

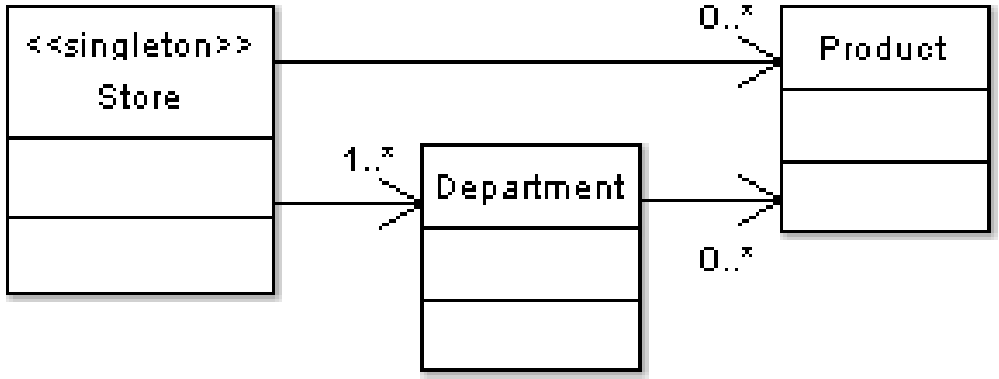
Learn how to cluster your Spring application declaratively and transparently with zero changes to existing application code

Agenda

- Overview of clustering solutions today
  - Sample problem statement
  - Traditional clustering solutions
  - Discussion: Scale-out OR Simplicity
- The need for Naturally Clustered Java™ technology
  - Scale-out AND Simplicity: Clustering at the Java VM (JVM™) level
- Introduction to the Terracotta for Spring
  - Overview of the Terracotta for Spring—features and demos
  - Summary and Q&A

Sample Application

- Our application
  - Inventory application (very naive app, but anyway…)
  - Spring based
- In-memory singleton consists of:
  - Product
  - Department
  - Inventory
  - Store



Sample Code: Product

```
public class Product {
  private double       m_price;
  private final String m_name;
  private final String m_sku;

  public Product(String name, double price, String sku) {
    m_name = name; m_price = price; m_sku = sku;
  }
  public synchronized void setPrice(double price) {
    m_price = price;
  }
  public synchronized double getPrice() { return m_price; }
  public String getName() { return m_name; }
  public String getSku() { return m_sku; }
}
```

Sample Code: Department

```
public class Department {
  private final String    m_code;
  private final String    m_name;
  private final Product[] m_products;

  public Department(
    String code, String name, Product[] products) {
    m_code = code; m_name = name; m_products = products;
  }

  public String getName() { return m_name; }
  public Product[] getProducts() { return m_products; }
}
```

Sample Code: Store Bean

```
public class Store {
  private final List m_departments = new ArrayList();
  private final Map  m_inventory   = new HashMap();
  public synchronized List getDepartments() {
    return m_departments;
  }
  public synchronized Map getInventory() {
    return m_inventory;
  }
  public synchronized void addDepartment(Department department) {
    m_departments.add(department);
  }
  public synchronized void addInventoryItem(
    String sku, Product product) {
    m_inventory.put(sku, product);
  }
}
```

Sample Code: Spring Bean Config File

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-
   beans.dtd">

<beans>
  <bean id="store" class="demo.inventory.Store"/>
</beans>
```

Problems

- Requirements
  - Need to enhance scalability
  - Need to ensure high-availability
  - Need to handle fail-over
- Solution
  - We need some sort of Clustering, e.g., sharing of state across many Java VMs

Problem Overview

- One Store per Java VM is simple—
  but does not scale
- Need to share the state across multiple nodes

- **How can we do it?**

Agenda

- Overview of clustering solutions today
  - Sample problem statement
  - Traditional clustering solutions
  - Discussion: Scale-out OR Simplicity
- The need for Naturally Clustered Java™ technology
  - Scale-out AND Simplicity: Clustering at the Java VM (JVM™) level
- Introduction to the Terracotta for Spring
  - Overview of the Terracotta for Spring—features and demos
  - Summary and Q&A

Solution 1: Java Message Service

- Use Publish-Subscribe (Topic)
- The `Store` has a JMS API Topic and subscribes on updates

- Note: Showing actual JMS code—can be simplified a little bit using Spring's `JmsTemplate`
- Note: Showing simplified code

Solution 1: Java Message Service

First We Need to Create Some Messages

```
private interface InventoryMessage
  extends Serializable {}

public class ProductMessage
  implements InventoryMessage {

  public static enum Type {CREATE, UPDATE, DELETE};
  private Product product;
  private Type    type;

  public ProductMessage(Product p, Type t) {
    product = p;
    type = t;
  }
  public Product getProduct() { return product; }
  public Type getType() { return type; }
}
```

Solution 1: Java Message Service

Then Add Setup to the Constructor for the Store Bean

```
...
InitialContext context = new InitialContext();
topicConnectionFactory = (TopicConnectionFactory)
  context.lookup(CONNECTION_FACTORY_JNDI_NAME);
topicConnection =
  topicConnectionFactory.createTopicConnection();
topicSession =
  topicConnection.createTopicSession(false,
  Session.AUTO_ACKNOWLEDGE);
topic = (Topic) context.lookup(TOPIC_NAME);
topicSubscriber = topicSession.createSubscriber(topic);
topicSubscriber.setMessageListener(this);
topicPublisher = topicSession.createPublisher(topic);
topicConnection.start();
...
```

Solution 1: Java Message Service

We Need CRUD-like Send Methods in the Store Bean

```
public synchronized void createProduct(Product product) {
  sendMessage(new ProductMessage(product,
  ProductMessage.Type.CREATE));
}
public synchronized void updateProduct(Product product) {
  sendMessage(new ProductMessage(product,
  ProductMessage.Type.UPDATE));
}
public synchronized void deleteProduct(Product product) {
  sendMessage(new ProductMessage(product,
  ProductMessage.Type.DELETE));
}
private void sendMessage(InventoryMessage msg) {
  try {
    Message message = topicSession.createObjectMessage(msg);
    topicPublisher.publish(message);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Solution 1: Java Message Service

Add the `MessageListener` Interface to the `Store` Bean

```
public class Store implements MessageListener {
  ...
  public void onMessage(Message msg) {
    try {
      if (msg instanceof ObjectMessage) {
        ObjectMessage objMsg = (ObjectMessage) msg;
        if (objMsg instanceof ProductMessage) {
          handleProductMessage(
              (ProductMessage) objMsg.getObject());
        }
      } else {
        ...
      }
    } catch (JMSException e) {
      ... // handle exception
    }
  }
  ...
}
```

Solution 1: Java Message Service

Finally We Need to Handle the `ProductMessage`

```
private void handleProductMessage(
  ProductMessage msg) {

  // check type (CREATE, UPDATE or DELETE)
  // perform action accordingly

  synchronized(this) {
    ... // implementation omitted
  }
}
```

Solution 1: Java Message Service

Problems

- JMS is asynchronous, but updates must be handled “synchronously”
- Potential “window” where other nodes might be out of sync (since messages take time to process)
- Concurrent modifications may take place and hard to handle
- Scalability and performance are terrible
  - Pub-Sub is a bottleneck
  - Serialization + marshalling and unmarshalling
- Extremely verbose code
  - Unnatural and error-prone

Solution 2: JCache

- Let's look at the standardization effort for distributed caching: JCache
- Basically a distributed HashMap
  - get() and put()

- Note: Using simplified code—omitting transaction management, etc.

Solution 2: JCache

Attempt 1—Course-grained Caching

```
// at startup time
cache.put("store", new Store());

// if we need to access Store info
Store store = (Store)cache.get("store");

// if we need to update a product in Store
Store store = (Store)cache.get("store");
store.getInventory().get("sku").setPrice(52.00);
cache.put("store", store);
```

Solution 2: JCache

Attempt 2—Fine-grained Caching

```
// only looking at how to handle Product now

// at startup time
cache.put("sku", new Product());

// if we need to access Product info
Product product = (Product)cache.get("sku");

// ok so far...but...
```

Solution 2: JCache

Updating the Product Is More Complex

- Now we need to maintain the **`Product-Department`** references ourselves

```
// if we need to update a product
Product product = (Product)cache.get("sku");
product.setPrice(52.00);
cache.put("sku", product);

// then we need a query mechanism to find the departments
Department[] deps = findDepartmentsWithProductID("sku");
for (int i; i < deps.length; i++) {
  // need to update all individual departments
  deps[i].getProduct("sku").setPrice(52.0);
  // need to put them back in cache
  cache.put(deps[i].getName(), deps[i]);
}
```

Solution 2: JCache

Problems

- Breaks Java technology's "pass-by-reference" semantics—developers need to maintain references manually
- Domain model is perturbed
- Adds unnatural, verbose, and error-prone coding rules
- Using serialization—impacts scalability
  - Can not keep track of actual changes
  - Flattens and sends whole object graphs over the wire

Agenda

- Overview of clustering solutions today
  - Sample problem statement
  - Traditional clustering solutions
  - Discussion: Scale-out OR Simplicity
- The need for Naturally Clustered Java™ technology
  - Scale-out AND Simplicity: Clustering at the Java VM (JVM™) level
- Introduction to the Terracotta for Spring
  - Overview of the Terracotta for Spring—features and demos
  - Summary and Q&A

Scale-out OR Simplicity: APIs Are Not Simple

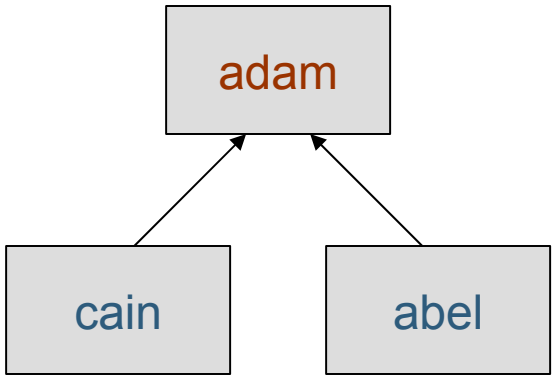
- Historically, clustering solutions **rely on Serialization**
- **This breaks object identity**
  - Data put into the cache and then read back will fail:

    `(obj == obj)` ⇒ `false`
- **Perturbs the Domain Model**
  - Management of object references using primary keys
- **Adds new coding rules**
  - Need to `get()` an instance, even if we already have a reference to it
  - Need to `put()` changes back—easy to forget
  - Can't trust callers outside the caching class to put a top-level object back in the cache if they edited it
- **Java technology should be simple…**

Problems With Serialization

Java Has “Pass-by-Reference” Semantics

```
// let's create one father and two sons
Person adam = new Person("Adam", null);
Person cain = new Person("Cain", adam);
Person abel = new Person("Abel", adam);
```

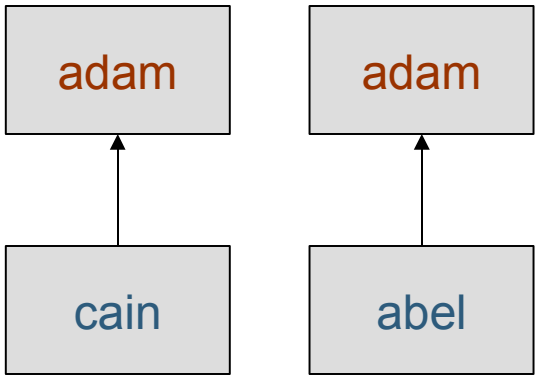


Object Identity **Is** Preserved

Problems With Serialization

Serialization Breaks Regular Object References

```
// but... if we serialize Cain and Abel
Person cain = (Person)Serializer.clone(_cain);
Person abel = (Person)Serializer.clone(_abel);
```



Object Identity **Is NOT** Preserved

The Importance of Preserving Java Technology's Pass-by-Reference Semantics

- If Object Identity is broken, then developers must:
  - Maintain the relational maps between objects themselves
  - Layer some kind of primary-key mechanism onto their domain objects
- This forces developers to:
  - Think like relational database designers
  - Rip the domain model apart and then manually stitch it back together with keys

API-based Clustering Is Not Scalable

- Java technology serialization is not scalable
- **Field updates**
  - ⇒ Push whole object graph
  - ⇒ Too much data is sent over wire
- **Coarse-grained locks**
  - ⇒ Locking top-level object, regardless of scope of change
  - ⇒ Premature lock contention

There Has to Be a Better Way!

- Let's take a step back and look at how Java technology works

Agenda

- Overview of clustering solutions today
  - Sample problem statement
  - Traditional clustering solutions
  - Discussion: Scale-out OR Simplicity
- The need for Naturally Clustered Java™ technology
  - Scale-out AND Simplicity: Clustering at the Java VM (JVM™) level
- Introduction to the Terracotta for Spring
  - Overview of the Terracotta for Spring—features and demos
  - Summary and Q&A

Ideally, Clustered Java Technology Would…

- Use natural Java code semantics
- Turn a single-Java VM application into a clustered one, **without**:
  1. Code changes
  2. Semantic changes

- What is needed is a Java-based service that handles these issues Transparently… at Runtime

Simplicity and Scale-out

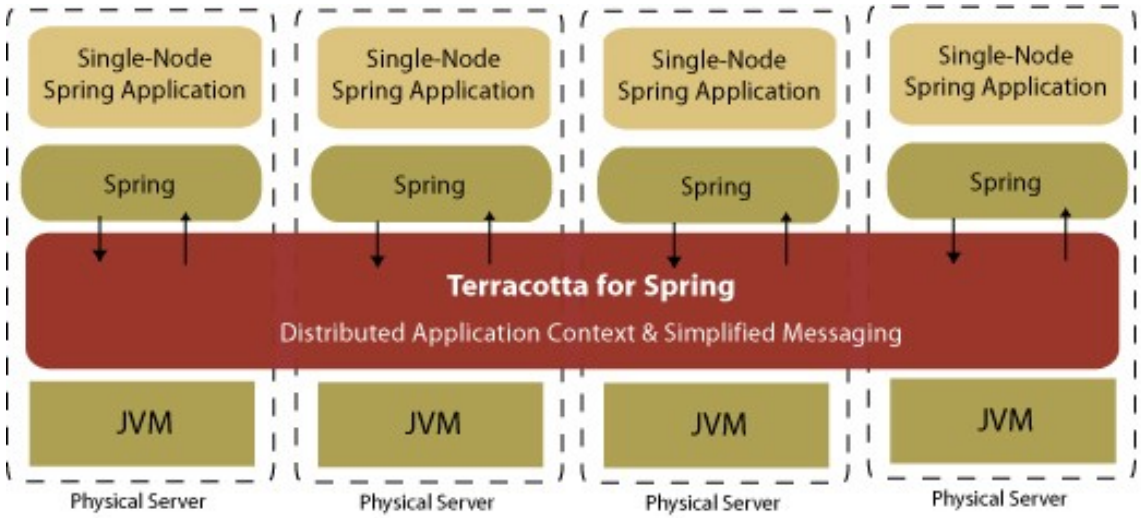
- Simplicity at runtime requires…
  - Preservation of Object Identity
  - Preservation of the semantics of the Java Memory Model
  - Event-based caching, not time-based
- Scale-out requires…
  - Fine-grained replication
  - Runtime lock optimization for clustering
  - Runtime caching for data access

Agenda

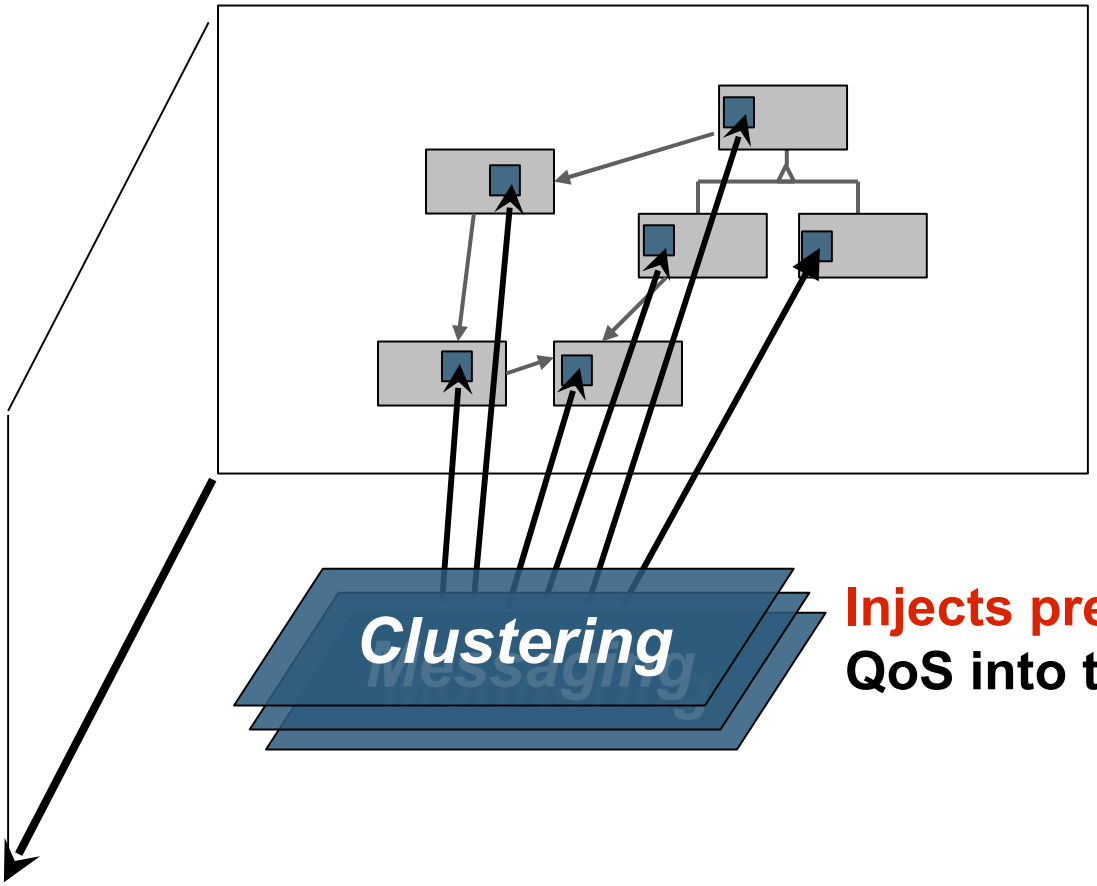
- Overview of clustering solutions today
  - Sample problem statement
  - Traditional clustering solutions
  - Discussion: Scale-out OR Simplicity
- The need for Naturally Clustered Java™ technology
  - Scale-out AND Simplicity: Clustering at the Java VM (JVM™) level
- Introduction to the Terracotta for Spring
  - Overview of the Terracotta for Spring—features and demos
  - Summary and Q&A

Clustering at the Java VM Level: Terracotta for Spring

Transparent Natural Runtime Clustering for Java Technology



Terracotta Injects Quality of Services Transparently at Runtime



Developer **focuses solely on the business logic**, using POJOs, Spring Beans, EJB™ specifications, etc.

Injects pre-packaged QoS into the application

Terracotta for Spring

The Spring Framework

- **Life-cycle**
  - Defines and drives object life cycle (creates and destroys beans)
- **Scope**
  - Singleton—scoped by application context
  - Prototype—scoped by user (factory returns a new one every time)
  - Session (or custom) scoped beans—scoped by session or custom code
- **Assembly**
  - Well-defined components with declarative dependencies
- Allows us to naturally layer clustering services on top

Introducing Terracotta for Spring

Overview

- **Drops In and Out** Transparently
- **Natural Clustering** of Spring Beans
- Turn Spring **`ApplicationContext`** Events into Distributed Reliable Events
- Sharing of Java Management Extensions (JMX) state
- Sharing of Spring WebFlow's page flows
- High performance—**fine-grained clustering**
- **Object identity is preserved**
- Cluster-wide **thread coordination**

Drops In and Out Transparently

- No changes to existing code necessary
- Declarative configuration in Terracotta XML file

Natural Clustering of Spring Beans

- Supported types are *Singleton* and *Session scoped* beans
- Life-cycle semantics preserved
- Scope semantics preserved—within the same "logical" **`ApplicationContext`**

DEMO

Clustered Inventory Spring Application

This Is All the Config We Need

```
<application name="Inventory">
  <application-contexts>
    <application-context>
      <paths>
        <path>*/inventory.xml</path>
      </paths>

      <beans>
        <bean name="store"/>
      </beans>
    </application-context>
  </application-contexts>
</application>
```

What Are Spring ApplicationContext Events?

- Spring has a simple event/messaging facility in the **`ApplicationContext`**
- Similar to the Observer pattern
  1. Publish event to the context using **`publishEvent(event)`**
  2. All beans that implement the **`ApplicationListener`** interface will receive the event

Distributed Reliable Events

- Turn Spring **`ApplicationContext`** events into Distributed Reliable Events
- Local within the same “logical” **`ApplicationContext`**
- Asynchronous and reliable multicast
- Highly performant
- Any POJO can be the event or part of the event
- No serialization—sends actual delta
- Pass-by-reference works as expected

Sharing JMX State

- Shared beans can be exposed through Spring JMX
- Coherent view of the aggregate state throughout the cluster
- One single point of management
- One single point of monitoring

Sharing of Spring WebFlow

- Clustering of WebFlow's state machine
- Transparent and high-performant **fail-over** for page flows
- Potentially allow sharing a WebFlow instance across an application, to be used by more than one user (when parallel tasks are required)

DEMO

Clustering JMX State—Web Application

Agenda

- Overview of clustering solutions today
  - Sample problem statement
  - Traditional clustering solutions
  - Discussion: Scale-out OR Simplicity
- The need for Naturally Clustered Java™ technology
  - Scale-out AND Simplicity: Clustering at the Java VM (JVM™) level
- Introduction to the Terracotta for Spring
  - Overview of the Terracotta for Spring—features and demos
  - Summary and Q&A

Summary

- Being able to Scale-out Spring applications is becoming more and more important
- Historically there has been a trade-off between Scale-out and Simplicity
- There is a need for a runtime that does not make this trade-off
- A runtime that can handle this:
  - Transparently—declarative config—zero code changes
  - While preserving the normal semantics for Java
- The Terracotta for Spring can address these issues today by clustering at the JVM level

Availability—Terracotta for Spring

- Free license for production use
- Sign-up for the beta program today
  - http://www.terracottatech.com/downloads.jsp

For More Information

- http://www.terracottatech.com/
- http://springframework.org/
- http://blog.terracottatech.com/
- http://jonasboner.com/

Q&A

Jonas Bonér

Transparently Clustered Spring—
A Runtime Solution for Java™ Technology

Jonas Bonér

Senior Software Engineer
Terracotta, Inc.

http://www.terracottatech.com

TS-3217