JBoss Certified Technology Partner

Israel JBoss User Group

Session 03 / 14.8.2006

JBoss and Spring: Integration Features

By : Zvika Markfeld, Tikal

Agenda

- Introduction
- JBoss Spring Deployer
- JBossMX and Spring JMX Support
- Spring Clustering with JBoss Cache
- Conclusion
- Resources

Introduction

Introduction to Spring

- Over the several several years, the Spring Framework has emerged as the de facto standard for developing **simple**, **flexible**, **easy-to-configure** Enterprise applications
- At the heart of Spring lies the ***Inversion of Control*** (IoC) design pattern:
  - Application as a set of POJOs
  - IoC Container wires them together, setting dependencies

Introduction to Spring – Cont'

- The Spring container is configured via a set of bean definitions, typically expressed via XML context files:

```
<bean id="MyServiceBean" class="mypackage.MyServiceImpl">
  <property name="otherService" ref="OtherServiceBean"/>
</bean>
```

- Client Code:

```
MyServiceInterface service =
  (MyServiceInterface)context.getBean("MyServiceBean");
service.doSomething();
```

Introduction to Spring – Cont'

- In addition to IoC, Spring provides many other services, coding conveniences, and "hooks" into standard APIs that ease server-side development
  - EJB(2.1, 3.0)
  - JMS
  - JMX
  - MVC frameworks
  - Transparent Remoting
  - And more…

Spring and EJB3

	EJB3	Spring
Dependency Injection	Coarse-grained, simple, JNDI-based	Fine-grained, versatile instantiation/lookup
Development Cycle	code -> compile -> package -> deploy -> test	short, environment-agnostic cycle
Transaction Propagation	In and across-Containers	In-Container only
AOP	Simple interceptor support	Full AOP stack
Transaction Manager	Container-provided	Pluggable
Persistence	JPA	JPA, Hibernate, JDO, etc.
Clustering	Non-mandatory, usually provided	Not provided, roll-your-own

Spring and EJB3 Integration

- Spring DAO Support:
  - JpaTemplate, JpaDaoSupport, JpaTransactionManager, etc.
  - EntityManager Injection via PostProcessor
  - Exception Translation
- Coarse-grained EJBs -> Fine grained Spring Services
  - EJB Container provides:
    - Clustering
    - Transaction propagation across JVMs
    - Classloading domains
  - Spring provides:
    - Everything else...
- The Pitchfork Project
  - WLS and Interface21 initiative, open source
  - Pluggable EJB3 container on top of Spring
  - Implements JSR220, JSR250

Spring Remoting Example

- A simple approach to Java-to-Java remoting in Spring is **HTTP Remoting**
- For example, the following context piece exposes MyService for public consumption (after servlet registration in web.xml):

```
<bean name="/MyRemoteService"
      class="org.springframework...HttpInvokerServiceExporter">
    <property name="service" ref="MyServiceBean"/>
    <property name="serviceInterface"
       value="mypackage.MyServiceInterface"/>
</bean>
```

- Actual service is injected into this bean definition and thus made available for the remote calls

Spring Remoting Example – Cont'

- On the client, the context definition reads:

```
<bean id="myServiceBean"
    class="org.springframework...HttpInvokerProxyFactoryBean">
 <property
   name="serviceUrl"
   value="http://somehost:8080/my-webapp/MyRemoteService"/>
 <property
   name="serviceInterface"
   value="mypackage.MyServiceInterface" />
</bean>
```

- Client-side code doesn't change!

```
MyServiceInterface service =
  (MyServiceInterface)context.getBean("MyServiceBean");
service.doSomething();
```

Spring Remoting – Up and Away…

- Besides HTTP, Spring supports exporting POJOs over several other remoting protocols, out of the box:
  - Web services
  - Hessian
  - Burlap
  - RMI
  - JMX(discussed later)
  - And others…

Spring and JBoss – a Match Made In Heaven?

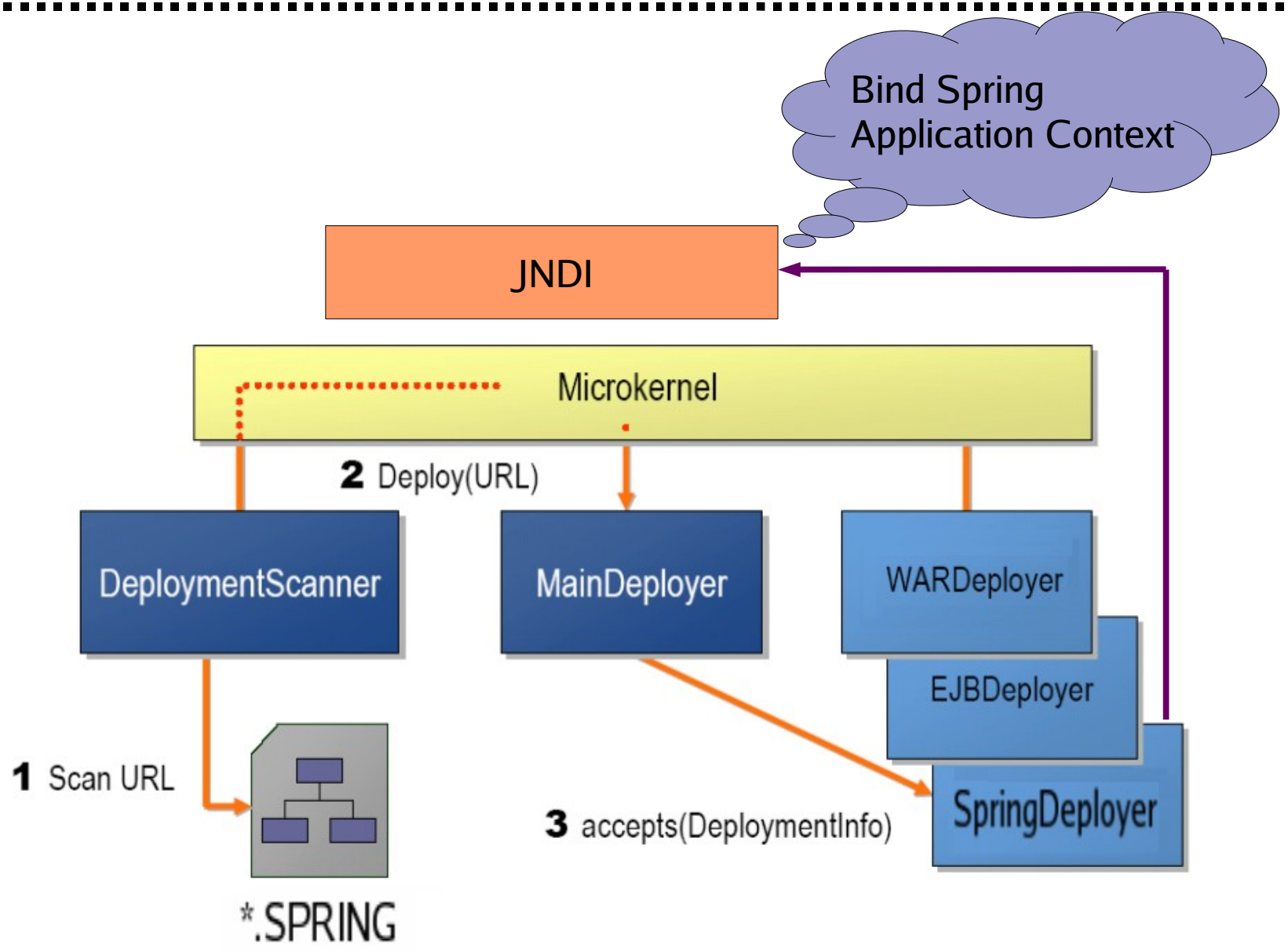
- Apart for the 'polical' disputes, it seems that there's a good match:
  - Advanced, innovative technology providers
  - Emphasize extensibility(each in its own domain)
  - Focus on the Enterprise Applications domain
  - Similar open source licensing
- Spring starts where JBoss stops:
  - Abstraction layers for common services
  - Standard, developer-friendly programming model
- In the next slides, we will examine some areas of integration…

JBoss Spring Deployer

JBoss Spring Deployer

- Allows deploying *.spring archives into JBoss
  - » Create JAR archive with a **META-INF/jboss-spring.xml**
- EJB 3.0 integration also supported
  - » Inject Spring-declared beans into an EJB using a **@Spring** annotation
  - » Based on JBoss 4's AOP capabilities
  - » Distributed as a JBoss 4 deployer archive
- Written by Ales Justin of jboss.org…

JBoss Spring Deployments Architecture



Spring Deployment Structure

```
my-app.spring/
   org/
      acme/
            ~MyBean.class
            ~MyBean2.class
   META-INF/
            jboss-spring.xml
```

Bean Classes

Standard Spring Configuration

Spring Annotations Example

```
@Stateless
@Interceptors(SpringLifecycleInterceptor.class)
public class RandomizerBean implements Randomizer {

    private WordsCreator wordsCreator;

    @Spring(jndiName = "my-app", bean="stateIntCreator")
    private IntCreator intCreator;

    public WordsCreator getWordsCreator() {
        return wordsCreator;
    }

    @Spring(jndiName = "my-app", bean="staticWordsCreator")
    public void setWordsCreator(WordsCreator wordsCreator) {
        this.wordsCreator = wordsCreator;
    }
}
```

Spring Archives Deployment

- The Deployer will register the XML defined bean factory into JNDI, in a non-serialized form
- Default JNDI name = short deployment filename (**my-app** in the example)
- Alternatively, drop the Spring XML file of the form **<name>-spring.xml** into the JBoss deploy directory
  - Put your jar libraries under **`server/<conf-name>/lib`**
- You can also embed these deployments inside an EAR, EJB-JAR, SAR, etc. as nested archives

Spring Deployment Metadata

- JNDI Naming and conetxt inheritence can be marked in jboss-spring.xml or *-spring.xml,
  as follows:

```
<beans>
      <description>
            BeanFactory=(<name>)
            ParentBeanFactory=(<parent name>)
      </description>
      ...
</beans>
```

JBossMX and Spring JMX Support

JBossMX and Spring JMX Support

- Spring JMX support provides 4 core features for transparent JMX integration, applicable for JBossMX:
  - Automatic Registration of any POJO as a JMX MBean
  - Flexible mechanism for controlling the management interface of your beans
  - Simple proxying of local and remote MBean resources
  - Declarative exposure of MBeans over remote, JSR-160 connectors

Exporting Spring Beans Over JBossMX

```
<beans>
  <bean id="testBean" class="com.mycompany.MyTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>
  <bean id="exporter"
      class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
       <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
  </bean>
  ...
</beans>
```

Exporting Spring Beans Over JBossMX – Cont'

- *testBean* is exposed as a JMX MBean under the ObjectName **bean:name=testBean1**
  - » Default ObjectName can be overrided by supplying a custom NamingStrategy object to the Exporter
- Bean's public properties and methods are exposed as MBean attributes and operations
- Lazy bean instantiation will be respected by the MBeanExporter, using a proxy

Automatic MBean Registration

- MBeans can be automatically detected by the MBeanExporter by setting the *autodetect* property to true:

```
<bean id="exporter"
    class="org.springframework.jmx.export.MBeanExporter">
  <property name="autodetect" value="true"/>
</bean>

<bean name="spring:mbean=true"
    class="com.mycompany.MyTestBean"/>
```

Bean should implement an interface ending with **MBean**

Fine-Tuning MBean Management Interface

- Behind the scenes, the MBeanExporter uses an *MBeanInfoAssembler* implementation for defining the exposed management interface of each bean
- The default implementation defines an interface that exposes all public properties and methods
- Spring provides two additional strategies for controlling the management interface:
  - Using source level metadata
    - Commons Attributes
    - Java5 Annotations
  - Using an arbitrary Java Interface

Using Java5 Annotation

```
@ManagedResource(objectName="bean:name=testBean4",
   log=true, logFile="jmx.log", currencyTimeLimit=15,
   persistPolicy="OnUpdate", persistPeriod=200,
   persistLocation="foo", persistName="bar")
public class AnnotationTestBean implements IJmxTestBean {
   private String name;

   @ManagedAttribute(currencyTimeLimit=20,
   defaultValue="bar", persistPolicy="OnUpdate")
   public void setName(String name) {
        this.name = name;
   }

   @ManagedOperation(description="Add Two Numbers Together")
   public int add(int x, int y) {
        return x + y;
   }
   ...
}
```

Annotation Configuration

```
<beans>
  <bean id="exporter"
      class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      ...
    </property>
    <property name="assembler" ref="assembler"/>
  </bean>
  <bean id="assembler“
      class="org.springframework.jmx...MetadataMBeanInfoAssembler">
    <property name="attributeSource" ref="attributeSource">
  </bean>

  <bean id="attributeSource"
      class="org.springframework.jmx...AnnotationJmxAttributeSource"/>
</beans>
```

Accessing MBeans via Proxies

- Spring JMX allows you to create proxies that re-route calls to MBeans registered in a local or remote MBeanServer
- These proxies provide you with a standard Java interface for interacting with MBeans

```
<bean id="proxy"
  class="org.springframework.jmx.access.MBeanProxyFactoryBean">

  <property
        name="objectName“ value=“bean:name=testBean“ />
  <property
        name="proxyInterface“ value=“com.mycompany.IMyTestBean” />
</bean>
```

Spring Clustering with JBoss Cache

Spring Clustering with JBoss Cache

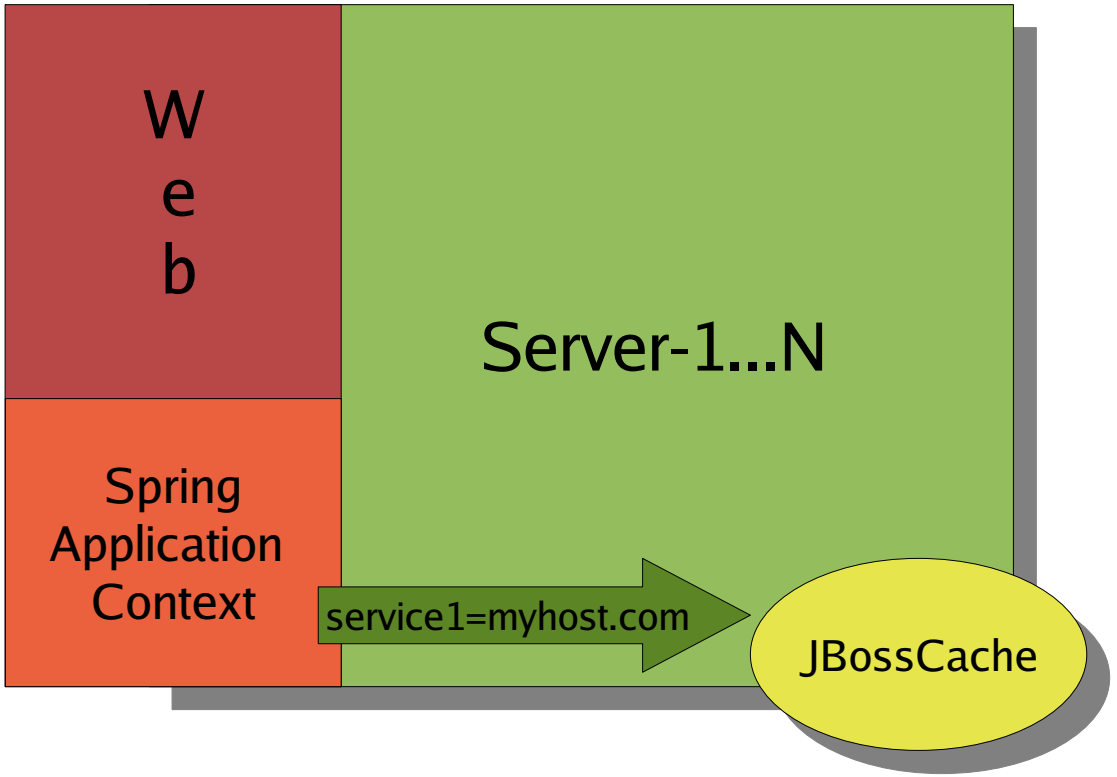
- **The domain:**
  Advertising pojos over http invokers
- **The problem:**
  URLs service identifiers are too brittle
  - » Server restarts, topology modification resistence
- **Possible solution:**
  Employ a naming service to provide dynamic, real-time resolution of service names
  - » Keep-alive based approach for maintaining online destinations: slow, non-transactional
- **…Yet better solution:**
  Distributed Cache
  - » Simple Map-like interface stores service endpoints

Distributed Cache as Dynamic Service Registry

- Associate a service name with one or more URLs pointing to its implementations
- Store the [name=(list of URLS)] associations in a distributed cache
- Update them as the network situation changes
- Service clients uses cache to inquire and access service implementations
- Bonus: client-side load balancing and failover
  - » Scalability better than in stateful session replication

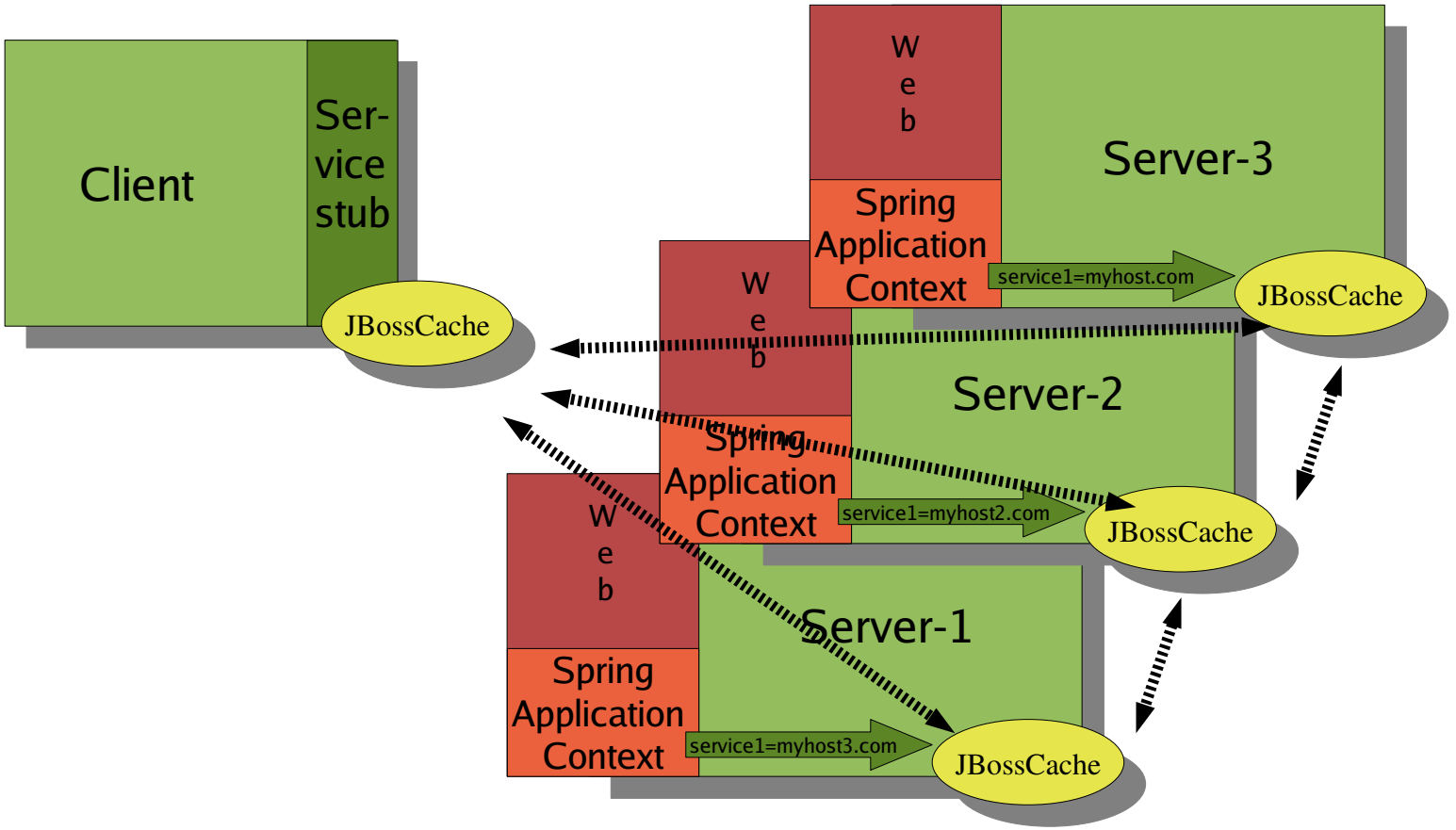
Suggested Topology

- Each server register the services it exports in the Cache, using the service's logical name as a key



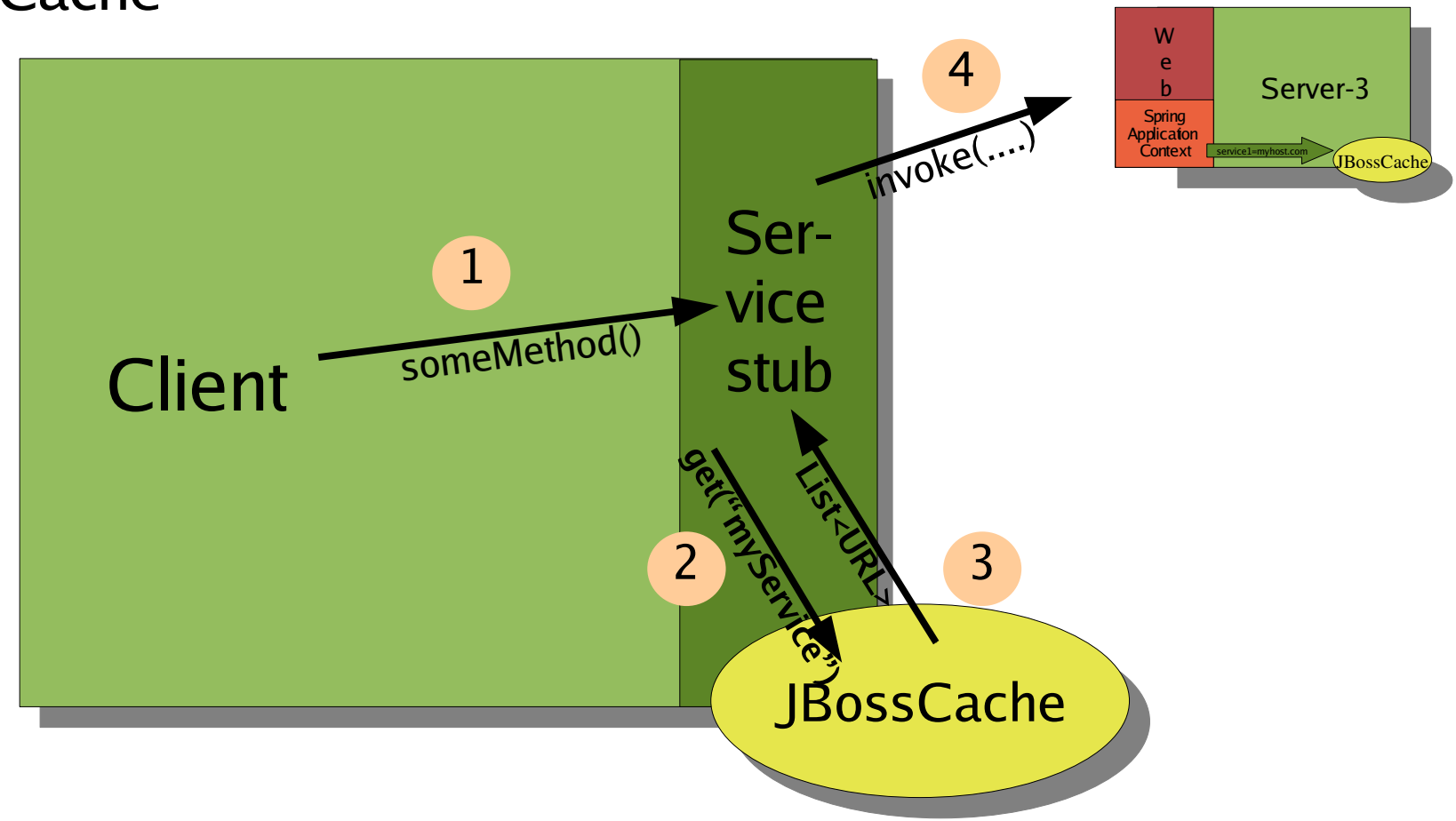
Suggested Topology – Cont'

- All participating nodes share the same cache, and so does the service client, using a cluster-aware stub



Suggested Topology – Cont'

- When the client needs to access the service, the smart stub finds an available endpoint by querying the local Cache



Setting Up Service Distribution

- **Step 1**: Exposing our TestService for remoting

```
<bean name="myService" class="com.mycompany.TestService"/>

<bean name="/TestService"
    class="org.springframework...HttpInvokerServiceExporter">
  <property name="service" ref="testService"/>
  <property name="serviceInterface"
    value="app.service.TestServiceInterface"/>
</bean>
```

Cache Proxy

- **Step 2**: Creating a Server-Side JMX Proxy for the JBoss Cache MBean

```
<bean id="customTreeCacheMBean"
    class="org.springframework.jmx.access.MBeanProxyFactoryBean">
  <property name="objectName">
    <value>jboss.cache:service=CustomTreeCache</value>
  </property>
  <property name="proxyInterface">
    <value>org.jboss.cache.TreeCacheMBean</value>
  </property>
</bean>
```

Cache Service

- **Step 3**: Writing an wrapper for the CacheMBean

```
public class JBossCacheServiceImpl
    implements CacheServiceInterface, InitializingBean {

  private TreeCacheMBean cacheMBean;

  public void put(String path, Object key, Object val)
      throws Exception {
    cacheMBean.put(path, key, value);
  }
  public Object get(String path, Object key) throws Exception {
    return cacheMBean.get(path, key);
  }
  public void setCacheMBean(TreeCacheMBean cacheMBean) {
    this.cacheMBean = cacheMBean;
  }
  public void afterPropertiesSet() throws Exception {
    cacheMBean.addTreeCacheListener(new MyCacheListener());
  }
}
```

Service Publisher

- **Step 4**: On each server node, a Spring Listener publishes and removes local services to/from Cache

```
private void contextRefreshed() throws Exception {
  logger.info("context refreshed");
  String[] names =
     context.getBeanNamesForType(HttpInvokerServiceExporter.class);
  logger.info("exporting services:" + names.length);
  for (int i = 0; i < names.length; i++) {
    String serviceUrl = makeUrl(names[i]);
    Set services = (Set)cache.get("/auto/svc/" + names[i], "KEY");
    if (services == null) {
      services = new HashSet();
    }
    services.add(serviceUrl);
    cache.put("/auto/svc/" + names[i], "KEY", services);
    logger.info("added:" + serviceUrl);
  }
}
```

Service Publisher Mechanics

- The publisher iterates over the list of services exported on http and adds their URLs to the cache
- The cache region, or path, contains the name of the service, whose URL list is stored as a **`java.util.Set`** under a fixed key in this region
- With JBoss Cache, it is important to make the service names part of the path to provide proper transactional scoping
  - » Updates made to different services will not interfere when they are mapped to different paths
    - /some/prefix/serviceA/key=(list of URLs)
    - /some/prefix/serviceB/key=(list of URLs)

Server-Side Configuration

```
<beans>
  <bean id="customTreeCacheMBean"
   class="org.springframework.jmx.access.MBeanProxyFactoryBean">
    <property name="objectName">
      <value>jboss.cache:service=CustomTreeCache</value>
    </property>
    <property name="proxyInterface">
      <value>org.jboss.cache.TreeCacheMBean</value>
    </property>
  </bean>

  <bean id="cacheService"
      class="app.service.JBossCacheServiceImpl">
    <property name="cacheMBean"
        ref="customTreeCacheMBean"/>
  </bean>

  <bean id="servicePublisher"
      class="app.service.ServicePublisher">
    <property name="cache" ref="cacheService"/>
  </bean>
</beans>
```

Client Local Cache Service

- **Step 5**: Writing an client-side local JBoss Cache delegate

```
public class LocalJBossCacheServiceImpl
     implements CacheServiceInterface, ApplicationListener {
  private TreeCache cache;

  public LocalJBossCacheServiceImpl() {
    cache = new TreeCache();
    PropertyConfigurator config =
        new PropertyConfigurator();
    config.configure(cache, "custom-cache-service.xml");
  }

  public void put(String path, Object key, Object val) {
    cache.put(path, key, value);
  }

  public Object get(String path, Object key) {
    return cache.get(path, key);
  }
```

Same Cache configuration as on server side (JGroups)

Local Cache Service – Cont'

```
    public void onApplicationEvent(ApplicationEvent event) {
      if (event instanceof ContextRefreshedEvent) {
                cache.startService();
      }
      else if (event instanceof ContextClosedEvent) {
                cache.stopService();
      }
    }
}
```

- The Client Service holds a reference to a **standalone** copy of JBoss Cache configured from the same configuration file used on the server
  - » JGroups communication layer definitions: UDP etc.
- All clients and servers access the same Cache!

Client Service Service Definition

```
<bean id="testService"
      class="com.mycompany.AutoDiscoveryProxyFactoryBean">
  <property name="serviceInterface"
    value="com.mycompany.TestServiceInterface" />
  <property name="cache" ref="localCacheService"/>
</bean>
```

- **AutoDiscoveryProxyFactoryBean**:
  - Written by app' developer once for all remote services
  - Extends **HttpInvokerProxyFactoryBean**
    - However, no *serviceUrl* provided
  - Automatically discovers services
    - Queries the local JBoss Cache
    - Query path is based the bean name
  - Automatically removes failed services
    - Updates the local JBoss Cache

Client Side Service Discovery

- **AutoDiscoveryProxyFactoryBean** obtains the list of URLs from our distributed cache:

```
private List getServiceUrls() throws Exception {
  Set services =
    (Set)cache.get("/auto/svc/" + beanName, "KEY");
  if (services == null) {
    return null;
  }
  ArrayList results = new ArrayList(services);
  Collections.shuffle(results);
  logger.info("shuffled:" + results);
  return results;
}
```

Summary

- Using the principles described in the last slides, you can deploy Spring Web Services that are:
  - Load-balanced between the members of your cluster
  - Stoppable, restartable with dynamic routing
  - Crash-resistent, failover-enabled

Resources

- *http://wiki.jboss.org/wiki/Wiki.jsp?page=JBossSpringIntegration*
- *http://static.springframework.org/spring/docs/1.2.x/reference/jmx.html*
- *http://www.javaworld.com/javaworld/jw-10-2005/jw-1031-spring_p.html*
- *http://www.jboss.org/products/jbosscache*

Q&A

Thank You

zvika@tikalk.com