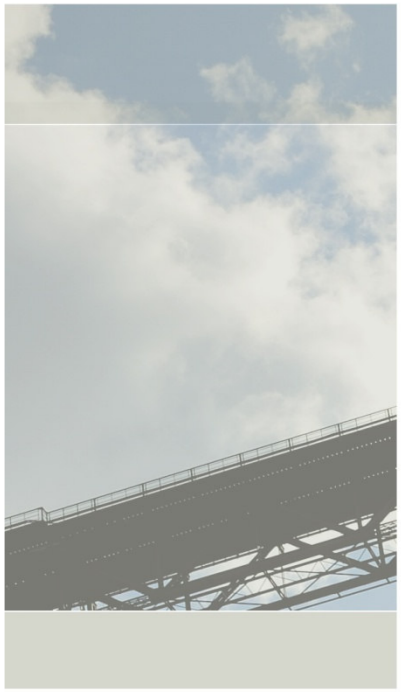
Advanced JBoss Cache

Carsten Mjartan
Senior IT Consultant
codecentric GmbH

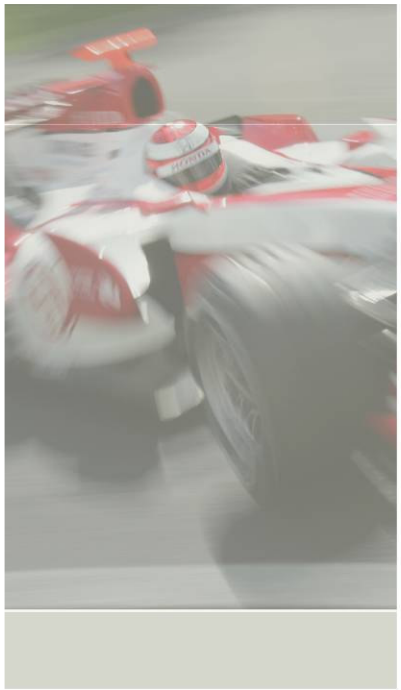
codecentric

Agenda

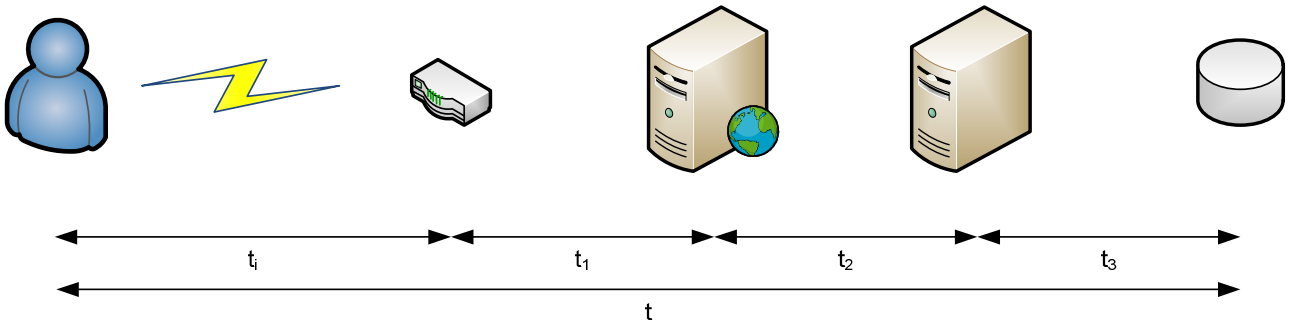


- Caching Grundlagen
- JBoss Cache Features
- Caching Einsatzmöglichkeiten

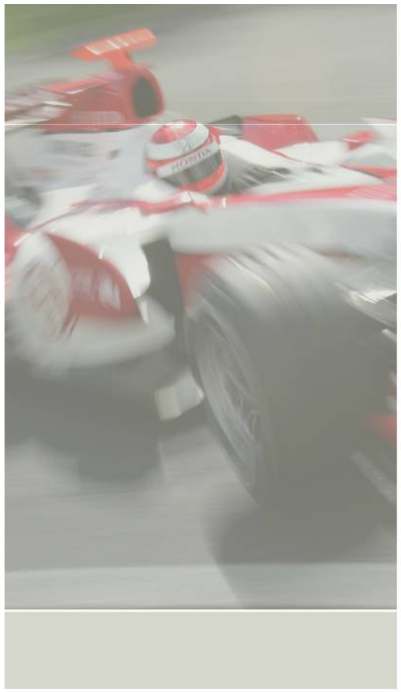
Caching



- Performante Zwischenspeicherung von Objektreferenzen
- Üblicherweise Nutzung zur Performance-Optimierung
  - Verringerung von Zugriffszeiten
  - Verringerung von benötigter Bandbreite
  - Entlastung von Backend-Komponenten



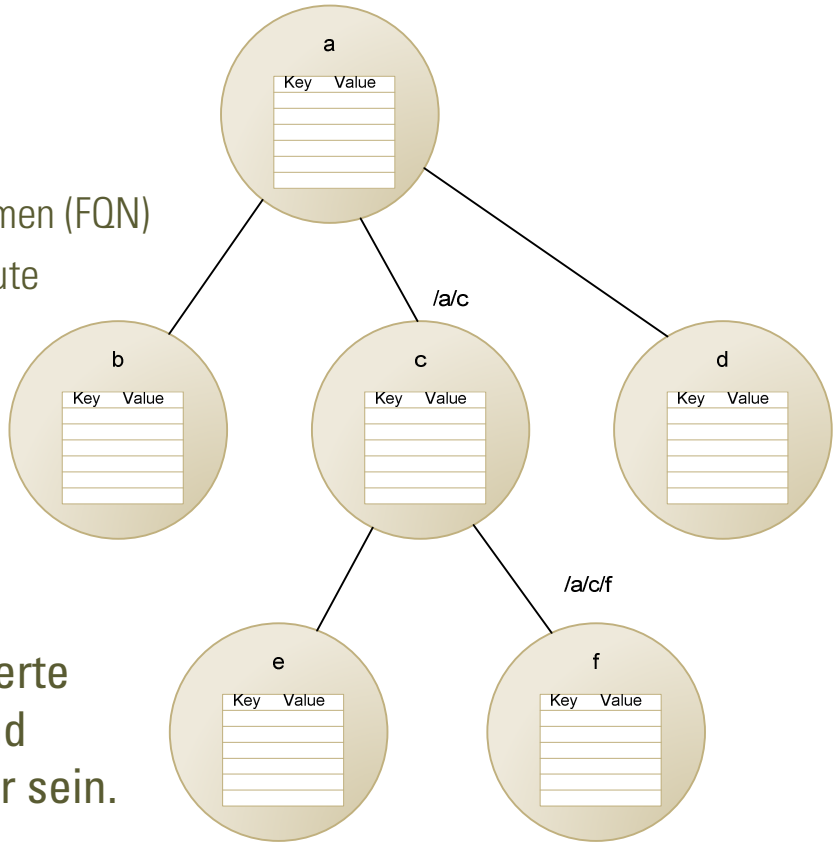
Caching (2)



- Nach Möglichkeit transparent für den Benutzer (nicht-invasiv)
- Im Regelfall Map-ähnliche API
  - Java JSR-107 (JCache) Cache Interface implementiert java.util.Map
- Optimal für
  - häufig benötigte, aber selten geänderte Daten
  - Daten, deren Änderung durch den Caching-Mechanismus verfolgt werden können
- JBoss TreeCache
  - Einfache Verwaltung von Daten in einer Baumstruktur
- JBoss PojoCache
  - Caching von Java-Objektstrukturen „Objektorientierter Cache“
  - Nutzung von aspektorientierter Programmierung (JBossAOP)

JBoss TreeCache-Struktur

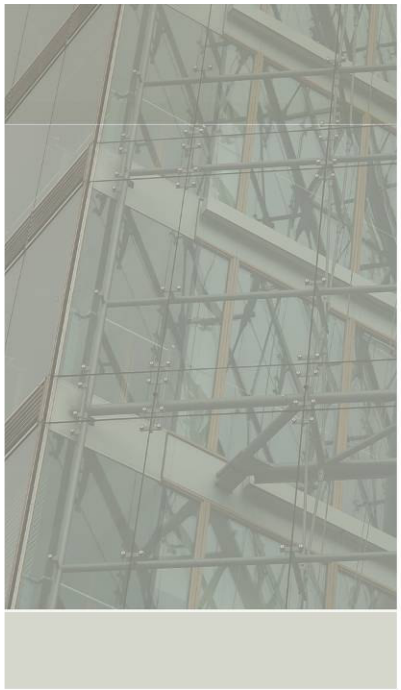
- Daten werden in einer Baumstruktur verwaltet
- Jeder Knoten besteht aus
  - dem Namen (Typ Object)
  - dem vollqualifizierten Namen (FQN)
  - einer HashMap für Attribute
  - evtl. Kindknoten
- Die Namen und Attributwerte müssen für Replikation und Auslagerung serialisierbar sein.



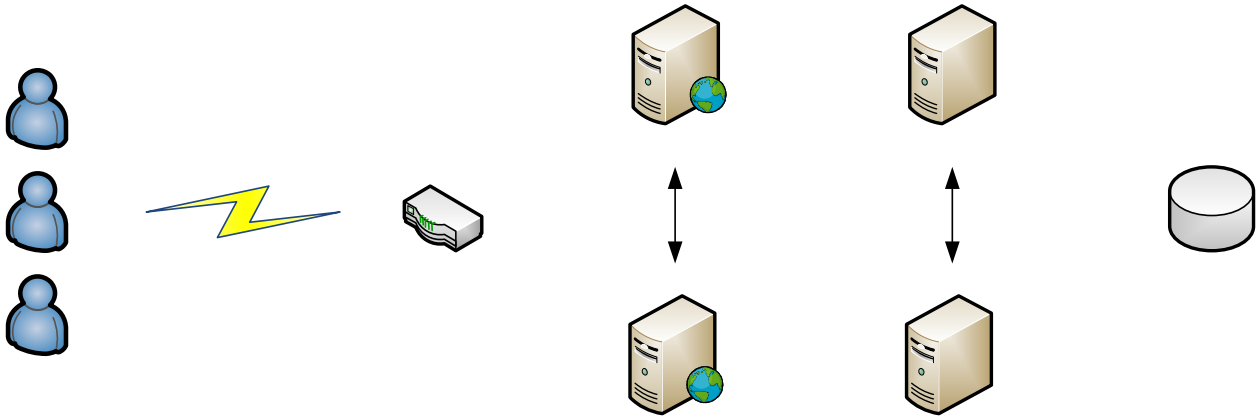
JBoss TreeCache-API

- Jeder Knoten wird über seinen vollqualifizierten Namen (FQN) eindeutig identifiziert.
  - `Fqn myFqn = Fqn.fromString("/a/c/f")`
  - `Fqn myFqn = new Fqn(new Object[] {"a", "c", "f" }`
- Die wichtigsten API-Methoden
  - `void put(Fqn name, Map data)`
  - `void put(Fqn name, Object key, Object value)`
  - `Object get(Fqn name, Object key)`
  - `Node get(Fqn name)`
  - `boolean exists(Fqn name)`
  - `void remove(Fqn name)`
  - `Object remove(Fqn name, Object key)`

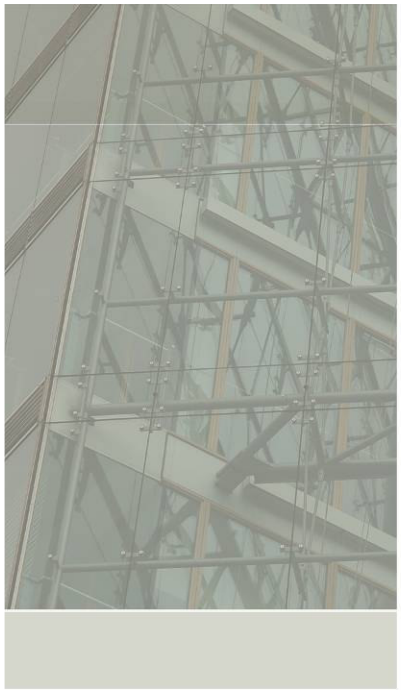
Verteiltes Caching



- Sicherstellung von Konsistenz bei verteilter Datenhaltung im Cluster
  - durch Replikation
  - durch Invalidierung
- Verbesserung der Skalierbarkeit (Load Balancing)
- Verbesserung der Verfügbarkeit

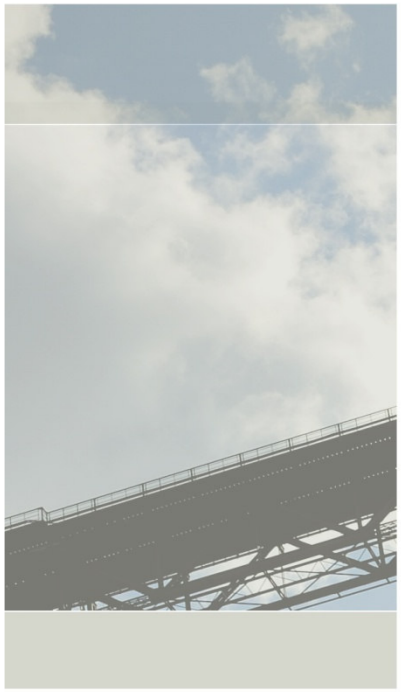


JBoss Cache Features



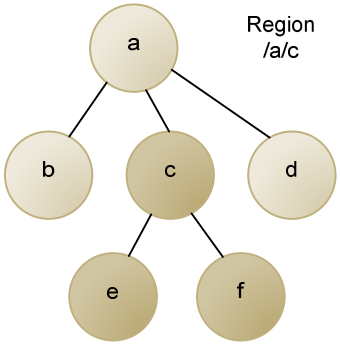
- Integration in JTA Transaktionskontext (z. Zt. nicht XA-fähig)
- Locking
  - Locking auf Transaktions- oder Thread-Ebene
  - Pessimistisches- und optimistisches Locking
  - Konfigurierbare Isolation Level
- Atomare Replikation
  - Replikation erst beim Commit
  - Gruppierung von Replikationsmeldungen
- Verteiltes Caching
  - Replikation (Buddy Replication) oder Invalidierung
  - Kommunikation über JGroups (TCP, UDP Multicast)
  - Synchrone / asynchrone Kommunikation

JBoss Cache Features - Eviction Policies



- Cache Größe ist begrenzt durch den dem Server zur Verfügung stehenden Arbeitsspeicher
- Cache-Einträge können veralten

- Eviction Policies regeln die Lebensdauer von Cache-Objekten, üblicherweise auf Basis von
  - Alter der Einträge
  - Anzahl der Einträge
  - Zugriffshäufigkeit
- Eviction Policies arbeiten auch im Cluster immer lokal
  - Keine Replikation notwendig
  - keine „verteilte“ Eviction
- Konfiguration:
  - Algorithmus (LRU, FIFO, …)
  - Parameter
  - **Basis: Cache Region**
    - Teilbaum im Cache

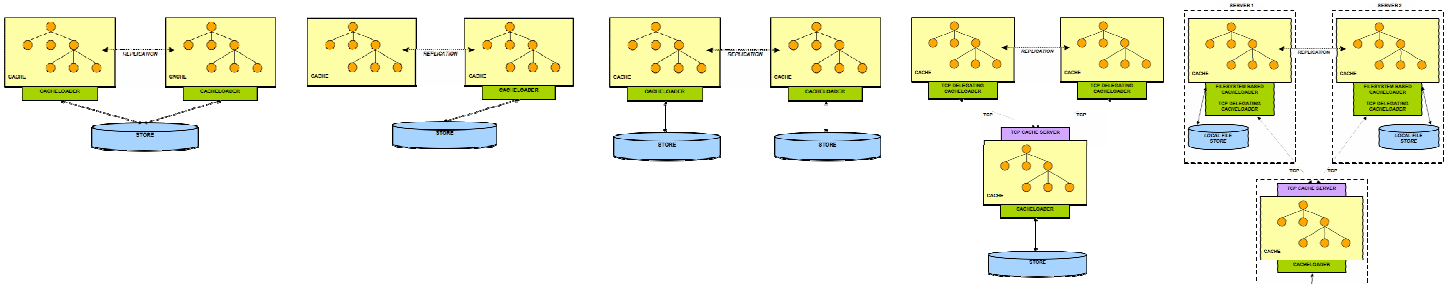


JBoss Cache Features - Cache Loader

- Speichern des Cache-Zustands in persistentem Datenspeicher
  - Dateisystem (File, JDBJE, JDBM)
  - JDBC
- Spezielle Cache Loader:
  - TcpDelegatingCacheLoader
  - ClusteredCacheLoader

- Lesen über Cache Loader, falls ein Knoten im Cache nicht existiert
- Schreiben über Cache Loader bei Erstellung / Modifikation eines Knotens
  - Konfiguration für Passivierung: Schreiben erst bei Eviction

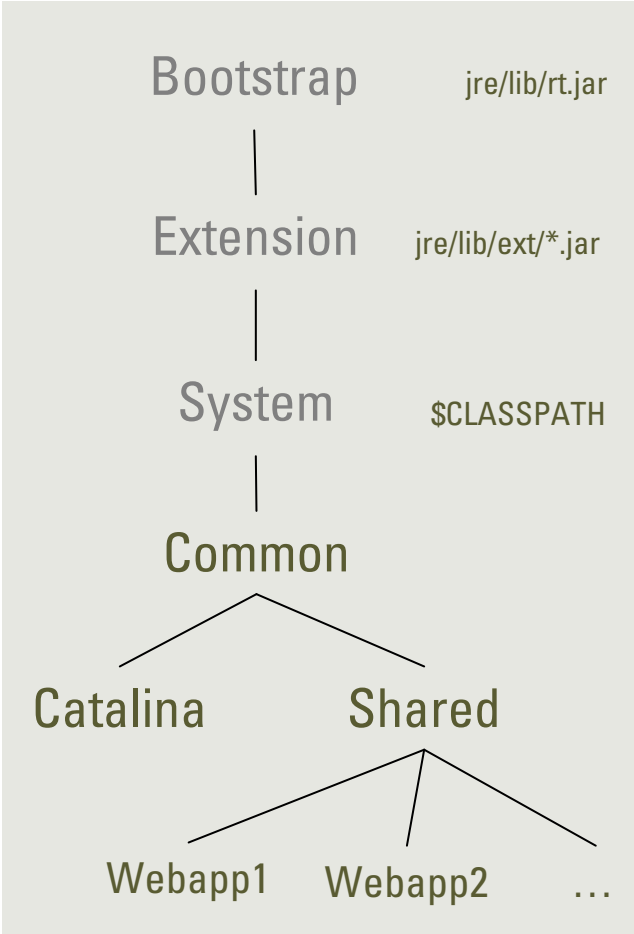
- Architektur abhängig von konkreten Anforderungen



codecentric

Region Based Marshalling

- Bei Replikation und Auslagerung von Daten über CacheLoader werden die im Cache gelagerten Daten serialisiert
- Für die Deserialisierung der Daten muss ein entsprechender ClassLoader existieren.
  - Bei Replikation auf Empfängerseite zum Zeitpunkt der Übertragung
  - Bei Zugriff auf im CacheLoader abgelegte Daten, z. B. während einer get()-Operation
- Thread ContextClassLoader nicht immer verfügbar (bspw. bei Deserialisierung)

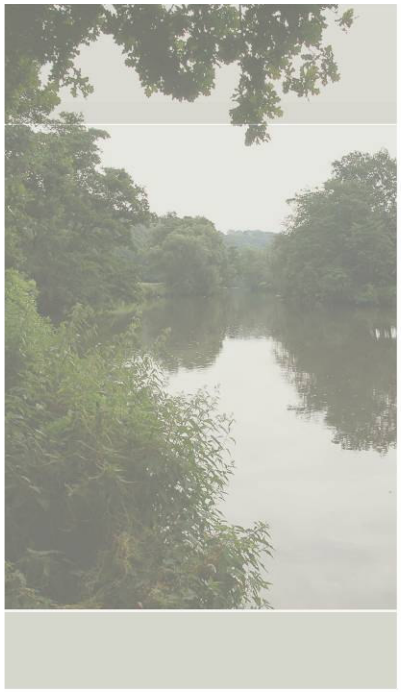


Tomcat 5.5 ClassLoader Hierarchie

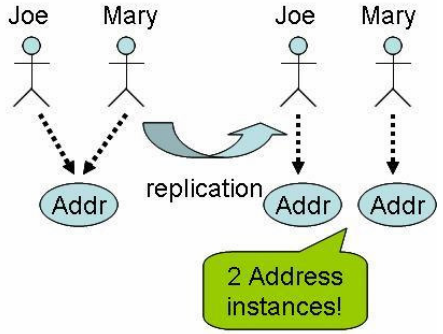
Region Based Marshalling (2)

- Registrieren eines Classloaders für eine Cache Region
- Aktivierung/Inaktivierung von Cache Regionen
  - per MBean im EAR oder per ServletContextListener
- In JBossCache 2.2 ist RegionBasedMarshalling deprecated
  - Neu: „Lazy Deserialization“
    - Reduktion des Deserialization-Overheads
    - Oben genannte Konfiguration nicht mehr notwendig
    - Etwas geringere Zugriffs-Performance

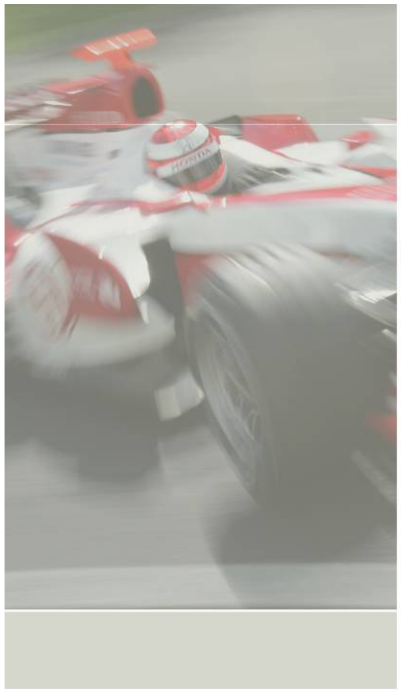
PojoCache



- Abbildung von Cache-Objekten auf Knoten und Attribute
- Zu speichernde Klassen müssen vorher um zusätzlichen Byte-Code erweitert werden
- Minimale Cache-API (hier: JBossCache 1.x)
  - `putObject(Fqn name, Object pojo)`
  - `Object getObject(Fqn name)`
  - `removeObject(Fqn name)`
- Löst einige Probleme herkömmlicher Caches
  - Performanter Umgang mit großer Cache-Objekte
    - Änderungen werden auf Objektebene repliziert/ausgelagert
    - bessere Skalierbarkeit durch feingranulares Locking
  - Cache-Objekte müssen nicht mehr Serializable sein
  - Lösung des Shared Instances Problem
  - Abbildbarkeit von Circular References



Caching Einsatzmöglichkeiten



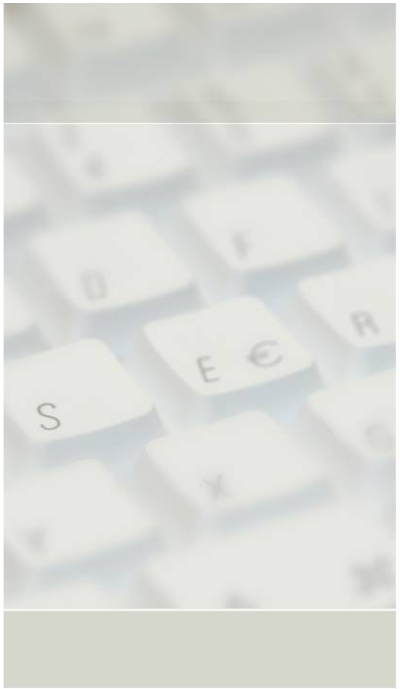
View/Controller	Business Logik	Datenzugriffsschicht
View Fragment Caching	Declarative Method Caching	Hibernate 2nd Level Cache
Session Caching/Replication	SFSB Clustering	Pojo Persistence
	Data Grid	

Aspekte

Security Cache

…und viele mehr

View Fragment Caching



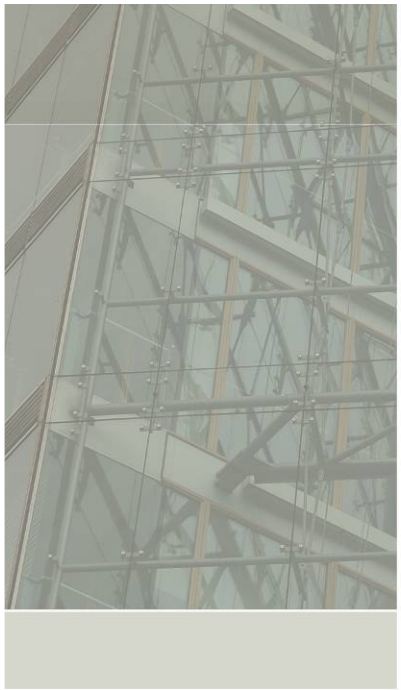
Zwischenspeichern von dynamisch gerenderten Bereichen im View

- bei sich selten ändernden, oft angezeigten Fragmenten
- bei komplexem Rendering oder
- bei teuren Data-Retrieval-Operationen (z. B. Lazy Loading im View)
- bei JSP, JSF und Templating Engines einsetzbar

- JSF Tag-Implementierung existiert in JBoss Seam
- JSP Tag Eigenimplementierung für JSP / Freemarker
- ggf. Feintuning durch <flush> Tag
  (wie in OSCache, Sun JSP Cache)

```
<s:cache key="recentEntries-#{blog.id}" region="welcomePageFragments">

  <h:dataTable value="#{blog.recentEntries}" var="blogEntry">
    <h:column>
        <h3>#{blogEntry.title}</h3>
        <div><s:formattedText value="#{blogEntry.body}" /></div>
    </h:column>
  </h:dataTable>

</s:cache>
```

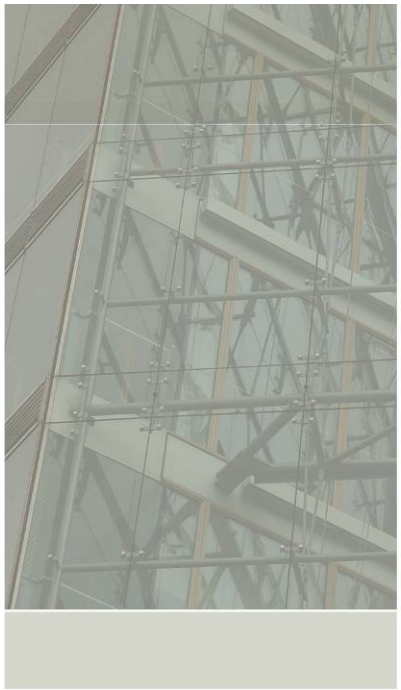
Session Caching/Replication



- Replikation von Session Attributen im Cluster
- JBossCache ist Grundlage für Session Replikation in JBoss 4/5
- Voraussetzung: <distributable/> Tag in web.xml

- Clustering Varianten
  - **Sticky Sessions**
    - Wechsel des Knotens nur bei Ausfall
    - Asynchrone Replikation für hohe Performance
    - Buddy Replication zur Sicherstellung von Skalierbarkeit
  - **Full Replication**
    - Verteilung von Requests auf alle Cluster-Knoten
    - Synchrone Replikation zur Sicherstellung von Aktualität
    - höchste Verfügbarkeit zum Preis geringerer Performance/Skalierbarkeit

Session Caching/Replication (2)



Granularität

- Session
  - Replikation der kompletten Session
  - Geeignet für kleine Sessions
- Attribute
  - Replikation von Einzelattributen
  - Geeignet bei größeren Sessions
  - Shared-Reference Problematik
- Field
  - Replikation per PojoCache bei Feldebene (Node-Replikation)
  - Geeignet für große Datenobjekte in Session
  - Byte-Code Enhancement der in der Session ablegbaren Klassen zur Compile/Deploy-Zeit notwendig

Replication Trigger

- SET
  - gute Optimierung
  - expliziter Aufruf von *setAttribute()* in Web-Applikation erforderlich
- GET
  - Dirty-Flag schon bei Lese-Zugriff
- SET_AND_NON_PRIMITIVE_GET
  - Default in JBoss AS
  - Dirty-Flag wird beim Lesen primitiver Datentypen nicht gesetzt
- ACCESS
  - Replikation bei jedem Zugriff und damit auch bei jedem Request
  - Last-Access-Time wird synchron gehalten

Declarative Method Caching

- Caching von Ergebnissen aus „teuren“ Methodenaufrufen
  - Daten- oder rechenintensive Services
  - Remote-Services
  - häufig aufgerufene Service (z. B. Entity Security)
- Annahme: stabiles E/A Verhalten
  ➔ Kombination von Eingangsparametern liefern konstantes Ergebnis
- Eingangsparameter:
  - Methodenparameter
  - ggf. Kontext (z. B. aktueller Benutzer)
- Realisierung als Aspekt!

Declarative Method Caching (2)

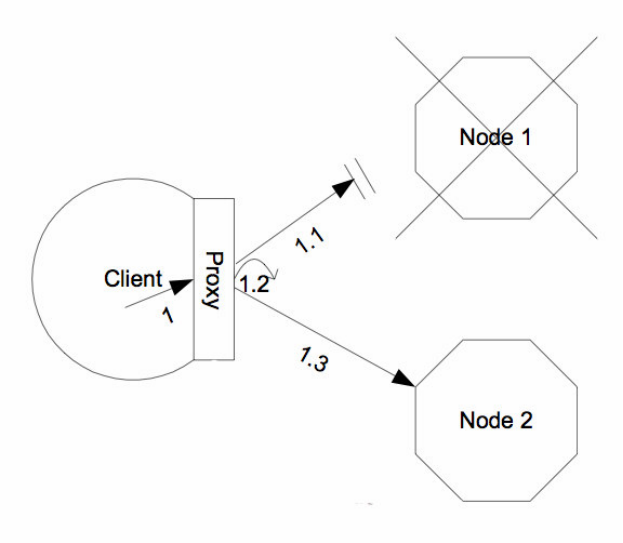
Zu beachten:

- Auslagerung und Replikation erfordern ggf. Marshalling
- Rückgabewerte sollten entweder immutable sein oder eine Kopie des Cache-Inhalts, um Nebeneffekte auszuschließen (Wrapping / Design gegen Interfaces)
- Bei Rückgabe von Hibernate Entities sollte stattdessen der 2nd Level Cache Mechanismus von Hibernate genutzt werden

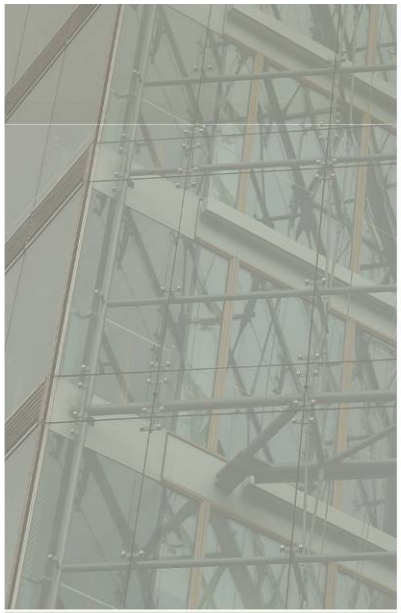
```
public class MyCacheableService implements CacheableService {

  @UserAware
  @Cacheable(modelId = "testCaching")
  public final String getName(int index) {
    // some implementation.
  }

  @UserAware
  @CacheFlush(modelId = "testFlushing")
  public final void updateName(int index, String name) {
    // some implementation.
  }
}
```

SFSB Clustering

- Replikation von Stateful Session Beans in einer Clusterumgebung
- Interne Verwendung von JBoss Cache in JBoss AS
- Aktuelle JBossCache Cluster-Member sind im SFSB Client-Proxy bekannt
  - hohe Verfügbarkeit
  - Zugriffsstrategie serverseitig konfigurierbar (Round-Robin, Session Affinity, …)



SFSB Clustering (2)

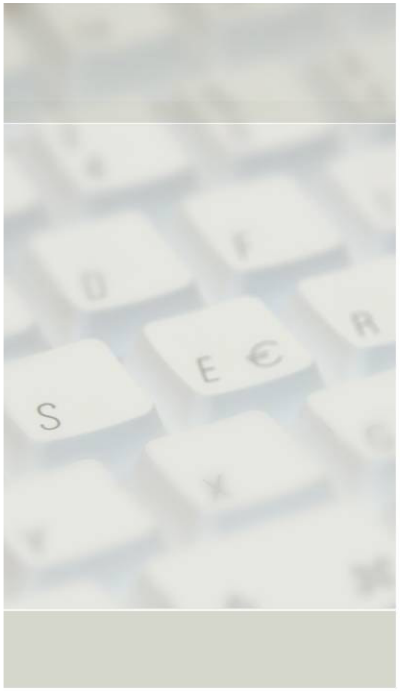


- JBoss Cache MBean Konfiguration anpassbar
- Eviction Policy Konfiguration kann über Annotationen erfolgen

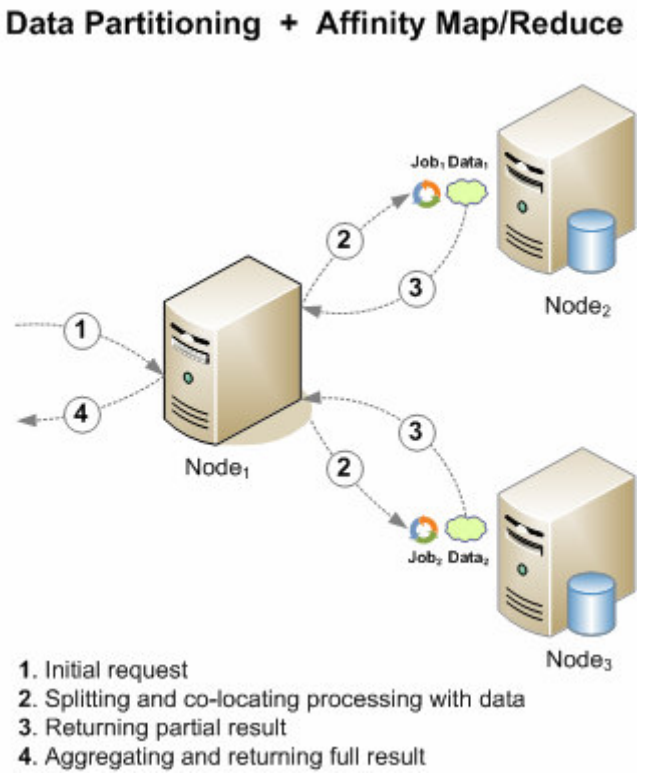
```
public @interface CacheConfig {
   String name()
      default "jboss.cache:service=EJB3SFSBClusteredCache";
   int maxSize() default 10000;
   long idleTimeoutSeconds() default 300;
   long removalTimeoutSeconds() default 0;
   boolean replicationIsPassivation() default true;
}
```

```
@Stateful
@Clustered
@CacheConfig(maxSize=5000,removalTimeoutSeconds=18000)
public class CounterBean implements Counter {

   private int state = 0;

   public void increment() {
      System.out.println("counter: " + (state++));
   }
}
```

Data Grid mit GridGain

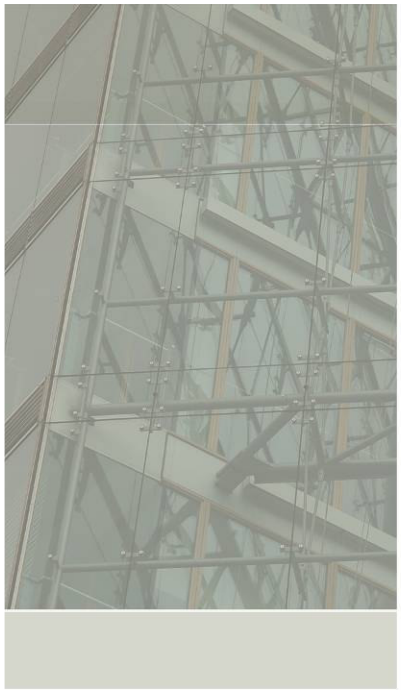


- Lokalität zwischen im Grid ausgeführten Jobs und den jeweils benötigten Daten
- Realisierung durch Kombination von *GridGain* und *JBossCache*
- *GridGain*: Open Source Grid Computing Implementierung für Java
  - Apache / LGPL Lizenz
  - Embeddable
  - Erweiterung von JBossCache um Data Partitioning



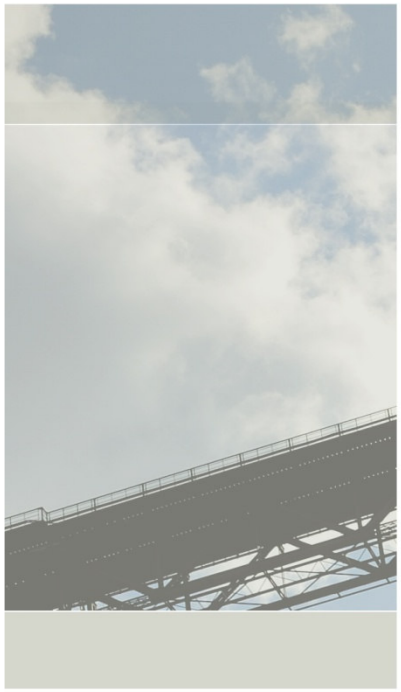
[von http://www.gridgainsystems.com]

Hibernate 2nd Level Caching



- JBossCache wird als Cache Provider für den Hibernate 2nd-Level Cache unterstützt
  - Caching von Entities, Collections und Query Results über Session-Grenzen hinweg
  - Eine mögliche Lösung für N+1-Problem
  - Transaktionales, verteiltes Caching
  - Replikation und Invalidierung (für Query Cache nur Replikation)
  - Ablage in Zwischenformat kein Marshalling erforderlich
- Hibernate / JBoss Cache Recommendations beachten: *http://wiki.jboss.org/wiki/JBossCacheHibernate*
- Cache Invalidierung mit Vorsicht zu verwenden
  - Invalidierung führt zu „Sticky Entities“
    - jedes Ablegen eines Entities im Cache führt zur Invalidierung auf allen anderen Knoten
  - Shared Cache Loader ohne Passivierung als Notlösung
  - Hibernate Cache Loader wäre wünschenswert
- Änderungen gecacheter Daten sollten ausschließlich über den DA-Layer erfolgen

Pojo Persistence

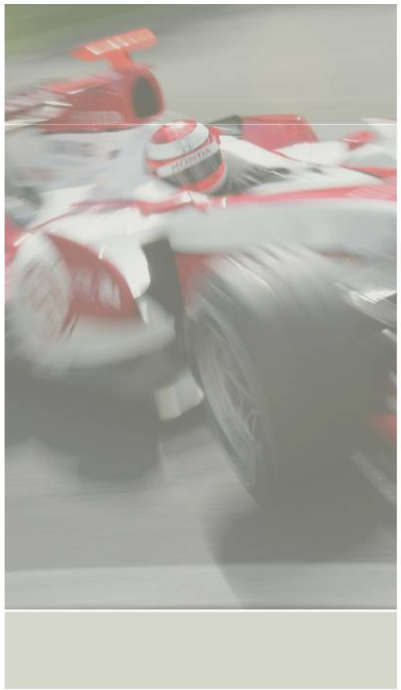


- Verwendung des PojoCache als einfache transparente Persistenzschicht

- Vorteile
  - hohe Performance
  - konfigurierbares Backend (in-memory, file-based, jdbc, …)
  - Transaktionalität, Locking, Distribution, …
  - einfache API

- Nachteile
  - proprietäres Speicherformat
  - keine Query-Language (hier denkbar: JXPath, Lucene, …)
  - Byte-Code Enhancement notwendig

Generelle Empfehlungen



- **Caching nicht als Selbstzweck**
  - Berücksichtigung bereits existierender Caches
    - Browser Cache
    - Datenbank-Cache
- **Überprüfen der Effektivität der Cache-Nutzung**
  - Performance Messung mit/ohne Cache
  - Messung mit realistischem Datenaufkommen / Userzahlen

codecentric GmbH

Grünewalder Str. 29-31
42657 Solingen

phone +49-212-2494315
fax +49-212-2494109
email info@codecentric.de
web www.codecentric.de

codecentric