# Hivemind
# Ein leichtgewichteter Container

*Manfred Wolff, wolff@manfred-wolff.de, www.manfred-wolff.de*

**Container sind Laufzeitumgebungen für Objekte. Der mächtigste Container im Java-Umfeld– der EJB Container – hat in der Entwicklergemeinde insbesondere in der Version 1.x und 2.x nicht durchgängig überzeugt. Als Gründe für die Ablehnung wird häufig unter anderem das komplizierte Programmiermodell genant. Dennoch kommt man in der Praxis nicht ohne eine Laufzeitumgebung für Geschäftsobjekte aus. Der Artikel schreibt den Jakarta Mikrokernel Hivemind.**

## Einleitung

Zunächst stellt sich allgemein die Frage, warum eine einmal entwickelte Geschäftslogik nur in einem bestimmten Container ablauffähig sein soll. Die Diskussion um POJO (plain old java objects) und POJI (plain old java interfaces) spielt dabei auch eine Rolle und soll später aufgegriffen werden.

Hinzu kommt, dass der EJB-Container für hochskalierende, verteilte Anwendungen konzipiert ist, die meisten Anwendungen aber keine verteilte Geschäftslogik besitzen. Meistens sind die verschiedenen Anwendungsbestandteile einer Anwendung auf mehreren Servern innerhalb des gleichen Netzwerksegments verteilt. Hier spielt vor allen Dingen die Frage der Skalierung und der Ausfallsicherung eine große Rolle. Die Verteilung der Geschäftslogik einer einheitlichen Anwendung auf verschiedene Server ist eher die Ausnahme, denn die Regel.

Der Verzicht auf einen Container, führt jedoch auch zum Verzicht auf eine Reihe von Add-Ons, die ohne Container vom Entwickler immer wieder selbst entwickelt werden müssen. Vor allem sind dies:

- Steuerung des Lebenszyklus von Objekten.
- Steuerung von intelligenten Mechanismen zur Wiederverwendung von Objekten wie Pooling oder Caching.
- Zugriff auf Ressourcen.
- Konfiguration von Komponenten.

Die Lücke schließen hier sogenannte leichtgewichtete Container. Bekannte Frameworks sind Spring, PicoContainer, Excalibur und HiveMind. Das in diesem Artikel HiveMind betrachtet wird soll die Mächtigkeit der anderen Frameworks nicht schmälern und liegt in den Präferenzen des Autors für Jakarta-Frameworks allgemein.

## HiveMind ein leichtgewichteter Container

HiveMind ist ein „kiss“ (keep it simple and stupid) und „easy to use“ Framework. Das Framework macht es einfach bereits existierende Komponenten zu „hiveminden“, also mit HiveMind zu verbinden. Andererseits fehlen auch Features wie z.B. „location transparency“, welches vom EJB Container angeboten wird. HiveMind Services beziehen sich immer lokal auf die gleiche JVM. In diesem Kapitel soll vor allem zwei Dinge beleuchtet werden:

1. Das Lifecycle-Management von Hivemind.
2. Das Erzeugen von Komponenten zur Laufzeit durch sog. dependency injection.

### POJO und POJI

Wie oben schon bemerkt gibt es eigentlich keinen Grund, warum eine einmal entwickelte Business-Komponente an eine bestimmte Laufzeitumgebung gekoppelt sein sollte. Die Lösung im EJB-Umfeld, um mit diesem Problem umzugehen, ist die Bereitstellung einer Business Facade. Die Business Facade benutzt nur ureigene Java Objekte und Java Interfaces. HiveMind arbeitet nur mit solchen Interfaces und Klassen, so dass eine Businsess Facade hier nicht notwendig wird.

### Das Livecycle Management von Hivemind

Für die Objekterzeugung bietet Hivemind vier verschiedene Modelle: primitive, singleton, threaded und pooled. Die ersten beiden Modelle bieten Singleton-Objekte an, wobei das Modell „singleton“ das lazy loading unterstützt. Das threaded-Modell erzeugt separate Objekte für jeden Thread und das vierte Modell bietet ein Pooling von Services.

| | Anwendungsweit | Threadbezogen | Lazy Loading |
|---|---|---|---|
| primitive | X | | |
| singleton | X | | X |
| threaded | | X | |
| pooled | | X | |

Mit dem dritten Modell ist es möglich zumindest für einen Request-Zyklus einen Status in der Komponente zu halten, während die anderen Modelle streng statuslos sind.

## In media res

Betrachten wir zunächst folgenden Usecase, das Login einer Anwendung:

- Die LoginAction ist eine Struts-Action die aufgerufen wird, wenn ein Login erfolgen soll.
- Der Benutzer hat die Möglichkeit einen neuen Account zu erstellen, in diesem Fall wird die AccountAction aktiviert.
- Die Login-Geschäftslogik bzw. der Usecase „Login“ ist ein einem Service gekapselt.
- Der Login-Service hat Zugriff auf eine Email-Komponente, um nach Erstellung eines Accounts eine Email mit einem generierten Paßwort zu verschicken.

- Der Email-Service nutzt wiederum einen Property-Service zur Konfiguration von Email-Host etc.

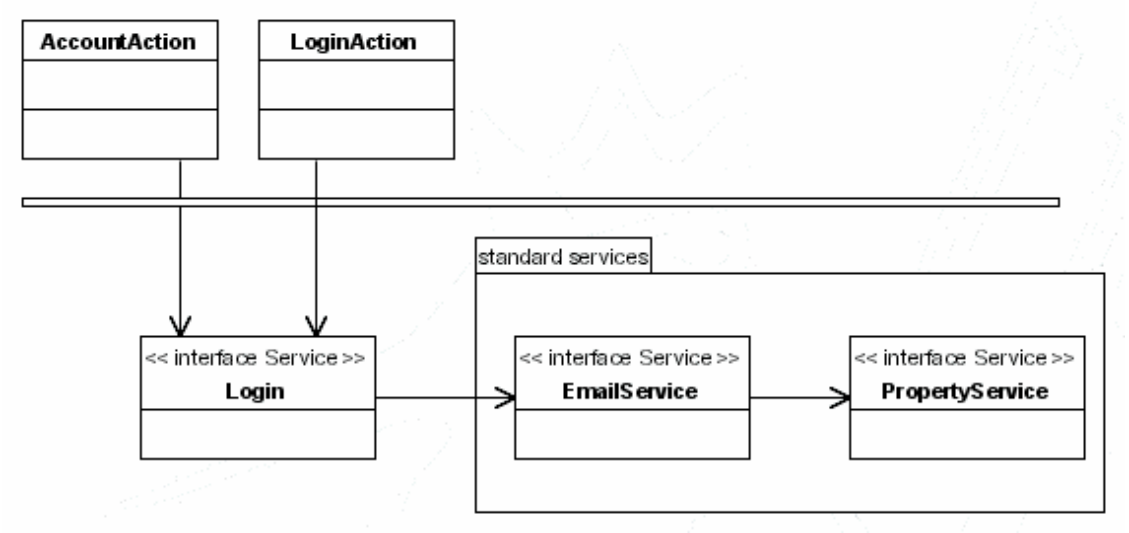


**Abbildung 1: Beispielarchitektur**

HiveMind soll nun folgende Aufgaben übernehmen:

1. Beim ersten Aufruf einer Methode des Login-Services soll dieser erzeugt werden und als Singleton der Anwendung zur Verfügung stehen.
2. Die Implementierungen des Email-Services und des Property-Services sollen mittels dependency injection zur Laufzeit beigesteuert werden.
3. Die beiden eher technischen Services sollen in einer separaten jar-Bibliothek als „standard-services“ zur Verfügung gestellt werden.

Eine Komponente als HiveMind-Komponente zur Verfügung zu stellen hat zwei Arbeitsschritte zur Folge:

1. Stelle die öffentliche Schnittstelle der Komponente als Interface zur Verfügung.
2. Stelle einen HiveMind-Deploymentdescriptor zur Verfügung, der pro Servicepoint das Interface mit einer konkreten Implementierung verbindet.

Interface und Implementierung sind ppjo (poji) leiten also nicht von irgendeiner Klasse eines Frameworks ab und sind auch ohne HiveMind ablauffähig und vor allem, was wichtig ist, ohne HiveMind testbar.

### Interfaces zur Verfügung stellen

Zunächst werden die Interfaces zur Verfügung gestellt. Am Beispiel des EMailService sieht das dann so aus:

```
public interface EmailService {

    /**
     * Sending a message. The parameters Hostname, EMailadress
     * and Sender are stored in a database.
     * @param emailAdress Email adress of the receiver
     * @param emailName Name of the email receiver
     * @param subject Subject of the email address
     * @param msg The Message.
     */
    void send(String emailAdress, String emailName, String
subject,
              String msg);
}
```

**Listing 1: Interface Deklarierung**

Es handelt sich also um ein ganz normales Interface. Auch die Implementierung ist ganz normal, als ob sie für ein normales pure Java Projekt getätigt wurde, außer die hervorgehobenen Zeilen:

```
public class EmailServiceDefault implements EmailService {

    /** Property Service */
    private PropertyService pservice;

    /**
     * Gets the configured property service
     * @return Returns the pservice.
     */
    public PropertyService getPservice() {
        return pservice;
    }

    /**
     * Sets the configured property service
     *
     * @param pservice The pservice to set.
     */
    public void setPservice(PropertyService pservice) {
        this.pservice = pservice;
    }
  ...
}
```

**Listing 2: Implementierung des Interfaces**

den hervorgehobenen Zeilen kündigt der EmailService an, dass er eine andere Komponente benutzen möchte. Er verlässt sich darauf, dass der zuständige Container die set-Methode aufruft und die Komponente bereit stellt.

```
    public void send(String emailAdress, String emailName,
                     String subject, String msg) {

        PropertyService service = getPservice();

        try {
            SimpleEmail mail = new SimpleEmail();
            mail.setHostName(getPservice().get("hostname"));
            mail.addTo(emailAdress, emailName);
            mail.setFrom(getPservice().get("mailfrom"),
                    getPservice().get("from"));
            mail.setSubject(subject);
            mail.setMsg(msg);
            mail.send();
        } catch (Exception e) {
            // @TODO
        }
    }
```

**Listing 3: Beispielsimplementierung**

### Hivemind Deployment Descriptoren

Der letzte Schritt ist HiveMind zu sagen, welche Implementierung für den PropertyServici zur Verfügung gestellt werden soll. Dies geschieht durch den Deployment Descriptor:

```
<module id="org.strutsit.examples.service" version="1.0.0">

 <service-point id="EmailService" interface=
    "org.strutsit.examples.service.EmailService">
    <invoke-factory>
       <construct
class="org.strutsit.examples.service.EmailServiceDefault"/>
     </invoke-factory>
    <interceptor service-id="hivemind.LoggingInterceptor"/>
 </service-point>

 <service-point id="PropertyService" interface=
    "org.strutsit.examples.service.PropertyService">
    <create-instance

class="org.strutsit.examples.service.PropertyServiceDefault"/>
    <interceptor service-id="hivemind.LoggingInterceptor"/>
  </service-point>

</module>
```

**Listing 4: Hivemind Modul Descriptor**

Man beachte die unterschiedlichen Einträge für den EMailService und den PropertyService:

- Services, die keine weiteren Services benötigen werden mit <create-instance> bekannt gemacht und bei Bedarf geladen.
- Services, die weitere Services benötigen werden als Factory deklariert und die Implementierung wird mit <invoke-factory> bekannt gemacht.

Die Vorteile einer solchen deklarativen Beschreibung liegen jetzt auf der Hand:

- Die Servcies sind POJOs, können also in einem JUnit Test einfach mit new() erzeugt werden.
- Sollen andere Implementierungen genutzt werden, z.B: ein Mock, so kann dies durch Eintrag in dem Deskriptor einfach geändert werden.
- Genauso kann der LifeCycle des Servcices einfach geändert werden:

```
<service-point id="EmailService" interface=
    "org.strutsit.examples.service.EmailService">
    <invoke-factory>
       <construct
class="org.strutsit.examples.service.EmailServiceDefault"
           model=pooled/>
     </invoke-factory>
    <interceptor service-id="hivemind.LoggingInterceptor"/>
 </service-point>
```

**Listing 5:**

In diesem Fall wurde das model von singleton (default) auf pooled geändert.

**Manfred Wolff** ist Diplom Informatiker und arbeitet seit fast 10 Jahren freiberuflich als IT-Dienstleister. Er konzipiert seit mehreren Jahren J2EE Projekte. Dabei setzt er bevorzugt Struts als Präsentations-Framework ein. Manfred Wolff ist Autor des Buches Struts ge-packt und vieler anderer Fachartikel.