<JSTORM>

EJB Home Factory

중앙대학교 컴퓨터공학과
자바 동호회
JSTORM
http://www.jstorm.pe.kr

Document Information

Document title:	EJB Home Factory
Document file name:	EJB_home_factory_ pattern_jstorm_1.0_final
Revision number:	<1.0>
Issued by:	<허경욱, 이용원>
Issue Date:	<2002/2/10>
Status:	final

Content Information

Audience	EJB 중급
Abstract	EJB 클라이언트는 EJB 홈 객체를 lookup해야 한다. 그러나, 같은 객체를 여러 번 lookup 하는 것은 중복이다. 그럼 어떻게 하나의 클라이언트가 그 애플리케이션의 생명주기 동안 오직 한번만 홈 객체를 lookup 할 수 있을까? 그리고 그 lookup 을 추상화 할 수 있을까?
Reference	http://www.theserverside.com/resources/patterns_review.jsp
Benchmark information	

Table of Contents

EJB Home Factory.....4

EJB Home Factory

EJB 클라이언트는 EJB 홈 객체를 lookup해야 한다. 그러나, 같은 객체를 여러 번 lookup 하는 것은 중복이다.

그럼 어떻게 하나의 클라이언트가 그 애플리케이션의 생명주기 동안 오직 한번만 홈 객체를 lookup 할 수 있을까? 그리고 그 lookup 을 추상화 할 수 있을까?

EJB의 리모트 인터페이스에 접근하기 위한 첫번째 과정은 EJB 홈 메소드의 JNDI lookup하는 것이다. 이 인터페이스에 접근하기 위해서 클라이언트는 반드시 InitialContext에 접근하고 InitialContext에 접근하는 도중에 EJB 홈을 형 변환하고 예외처리하는 실제적인 lookup을 수행하는 집중적이고 비싼 과정을 거쳐야 된다.

```
try {   // initial context 를 얻기 위해서

  Properties properties = new Properties();

  // 네임 서비스의 URL을 얻는다.
  properties.put(javax.naming.Context.PROVIDER_URl,
      "some providers url");

  // initial context factory 의 이름을 얻는다
  properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
       "some name service");

  initContext = new InitialContext(properties);
} catch (Exception e) {
  // initial context를 얻는 동안 발생하는 예외 ...
}
```

```
try { // JNDI 네임을 사용하는 홈 인터페이스를 lookup 하기 위해

  Object homeObject = initContext.lookup("aHomeName");
  myHome = (MyHome)
      javax.rmi.PortableRemoteObject.narrow(homeObject,
      MyHome.class);
} catch (Exception e) {
  // 홈 인터페이스를 얻는 동안 발생하는 예외 ...
}

// EJBObject 스텁을 얻는다.
MyEJB anEJB = myHome.create();
```

위 코드의 예는 EJB 홈 lookup 이 얼마나 복잡하고 지겹고 많은 반복을 할 수 있는지 보여준다. 문제는 한 특정 프로그램이 많은 EJB 홈 참조들의 사용을 한다는데 있다. (EJB 홈 참조는 한 클라이언트가 참조할 필요가 있는 각 EJB 중 하나이다.) 그리므로, 각 EJB 홈을 lookup 하기위한 코드를 쓰는것은 정말로 중복된 코드이다.

최악의 경우는 홈을 찾은 후 한번 사용한뒤 다시 사용하지 않는 경우이다. EJBHome 을 찾기 위한 JNDI lookup 을 매번하는 것은 많은 비용이 들 수 있다. 예를 들어 보면

① JNDI 서버가 다른 머신에 있을 경우에는 네트워크 호출이 필요하다.

클라이언트가 JNDI 서버와 다른 자바 가상 머신에 있을 경우 JNDI 호출은 네트워크 호출을 필요로 할 것이다. 이런 것이 발생하는 경우는 EJB 서버의 JSP/서블릿 서버가 다른 자바 가상 머신에 있고 JNDI 서버는 EJB 서버와 같이 있는 클러스터 환경이 한 예이다.

② JNDI 서버와 클라이언트가 같은 머신에서 구동되어도 IPC(Inter Process Communication)가 필요할 수 있다.

클라이언트가 EJB 서버와 같은 컴퓨터에 있어도 같은 자바 가상 머신에서 구동되지 않으면 EJBHome을 찾는데 IPC 오버헤드가 있다.

클라이언트가 EJB 서버(JNDI 서버을 포함하는) 와 같은 가상 머신에 구동된다 할지라도 매번의 웹의 리퀘스트마다 EJB 홈을 찾는 경우에 EJB 홈은 클라이언트의 생명주기동안 수정되지 않고 재사용될 수 있기 때문에 효율에 나쁜 영향을 끼친다.

특정 페이지 뷰가 분당 500번 이상 페이지 뷰가 일어날수 있는 높은 트래픽의 웹 사이트를 상상해 보자. 500개의 다른 클라이언트로부터 500번의 같은 객체를 찾는 퍼포먼스 오버헤드는 많이 생각해 줘야 될 정도로 크다. 그리고 완전히 불 필요한 것이다.

EJB 홈을 찾는데 나은 방법은 먼저 lookup 코드를 추상화하고 클라이언트 생명주기동안 같은 EJB 홈 인스탄스는 재사용하는 것이다. 그러므로, EJB 홈을 찾는 코드는 재사용 가능한 “EJB Home Factory”로 추상화하고 “EJB Home Factory”는 클라이언트의 생명주기동안 EJB 홈 인스턴스를 캐시 할 수 있게 한다.

“EJB Home Factory”는 그림 1에 나온것처럼 싱글튼 패턴(Singleton Pattern)으로 구현된 일반 자바 클래스이다. 팩터리는 EJB 홈을 찾는 로직을(어떤 타입의 EJB 홈이라도 재사용가능하게 로직을 만든다) 캡슐화 하고, 두번째 검색 요청마다 캐시된 홈을 전달할수 있게끔 내부적으로 홈을 캐시한다. “EJB Home Factory”는 어떤 애플리케이션에서도 재사용이 가능하다. 이것은 getXXXHome 형식의 특정 영역에 한정된 lookup 코드를 포함하지 않고, 대신에 lookup 홈 메소드를 사용하면서 가능하다. 이 팩터리는 또한 애플릿에서 서블릿까지 또는 EJB들 간이라도 어떤 환경에서도 사용가능하다. 이는 EJB 홈 찾는 것을 최적화 하고 캡슐화 하였기 때문이다.

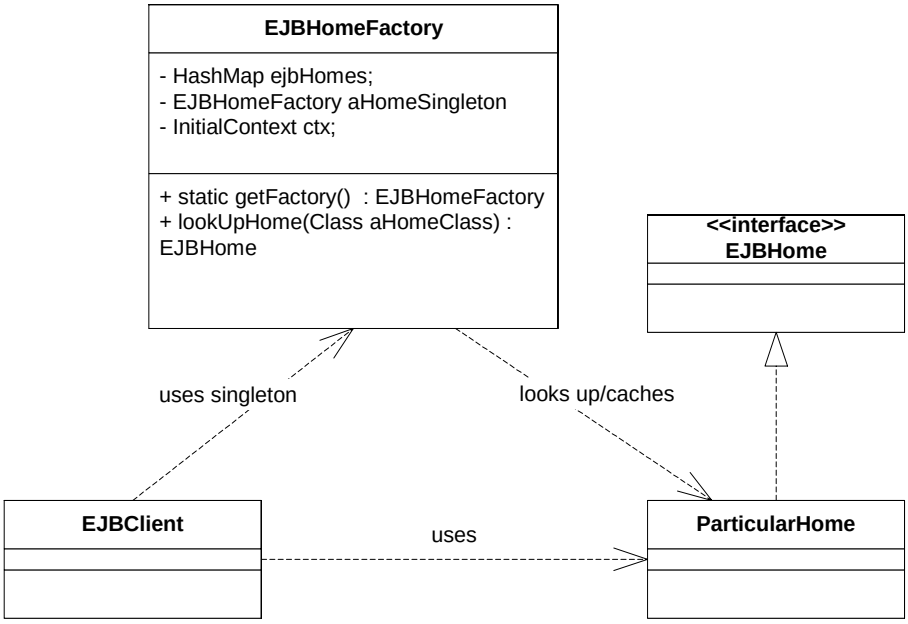

그림 2. EJBHomeFactory

이 팩터리의 간단한 구현은 이 문서의 마지막에 있는 예제코드1 에 있다. “EJB Home Factory”의 사용법은 간단하다. 클라이언트에선 lookup 코드를 한 줄로 줄이면서 홈 lookup에서 생성 로직까지 완전히 추상화 되었다. 예를 들어 Account bean 클라이언트는 아래 코드를 수행할 것이다.(예외처리부분은 간단한 이해를 위해 제거되었다.)

```
AccountHome anAccountHome =
(AccountHome)EJBHomeFactory.getFactory().lookUpHome(AccountHome.class);
```

기억하라! 클라이언트는 account 홈 인터페이스에 JNDI 네임 대신에 .class 형식으로 전달해야 된다. “EJB Home Factory”를 사용하기 위해선 클라이언트는 EJB의 JNDI 네임 보다 더욱 추상화 된다. 클라이언트가 알아야 될 전부는 요청시 필요한 홈 객체의 메소드 인터페이스이다. (파라미터로 넘어갈 .class의 이름과 리턴되어 나오는 EJB 홈의 형태) 그러나, 클라이언트 는 홈 객체의 메소드를 호출하기 위해선 어떻게든 이 메소드 인터페이스를 사용해야 된다.

클라이언트가 특정 홈 객체를 위해서 “EJB Home Factory”를 처음 호출할 때 이 팩터리는 JNDI를 통해 찾을 것 이다. 그리고, 그 홈 객체를 hashmap 에 내부적으로 캐시한다. 다음 호출부터는 이 팩터리는 완전히 홈 호출을 최적화해서 캐시된 EJB 홈의 복사본을 전달할 것이다.

여기에 이 패턴과 간단한 구현을 위하여 약간의 구현상 요구되어야 되는 조항이 있다.

① “JNDI Environment Entry” 는 외부 디스크립터에 재분해된다.

코드내에 일일이 손수 JNDI 팩터리와 URL 문자열을 properties 객체에 넣고 그것을 InitialContext 생성자에 전달하는 것 보다 (첫번째 코드예처럼) 그런 properties을 클라이언트 디플로이먼트 디스크립터나 runtime system propertie에 구체화해야 한다. (웹 애플리케이션은 web.xml에, ejb일 경우는 ejb-jar.xml에, 독립 실행형 클라이언트일 경우는 –D flag로... etc) 이 경우에 InitialContext 객체는 런타임 시 system properties에서 자동으로 properties을 찾을수 있다. 이것은 팩터리 구현을 간단하게 만들고 더욱 재사용가능하게 만든다.

② JNDI 네임 매핑하는 방법이 따로 있어야 될 경우가 있다.

예제코드1의 매핑은 JNDI 네임이 완전히 class 네임에 부합한다. (ie: com.xxx.xxxHome) 이런 네임 매핑이 불가능한 경우에 JNDI 네임 매핑은 외부 디스크립터에 정의되어 있어야 한다. 팩터리 생성자에서 이런 environment entries는 .class에서 JNDI 네임으로 일치되어야 하고 내부적으로 다른 hashmap에 저장되어야 한다. 다른 방법은 JNDI 네임을 같이 전달하는 수가 있다.

(lookUpHome(class이름) 을 lookUpHome(JNDI이름, class이름)로)

“EJB Home Factory” 패턴은 완전히 일반적이고 재사용가능한 형태이고 또한 클라이언트의 EJB 홈 lookup 하는 복잡도를 추상화 시키는 간단하고 효과적인 방법이다. EJB 홈을 캐시함으로 해서 효율은 눈에 띄게 증가된다. 이것은 홈을 찾는 중복을 제거함으로 나타나는 것이다. “EJB Home Factory”는 일관성있는 홈 객체 인터페이스를 제공한다. 그러므로 어떤 환경(애플릿, 서블릿, 독립 실행형 애플리케이션, EJB들간...)에서나 재사용가능하다.

예제 프로그램

```
import javax.ejb.*;
import java.rmi.*;
import java.util.*;
import javax.naming.*;

/**
 * “EJB Home Factory”,
 * EJBHome 의  간단한 hashmap 캐시를 유지한다.
 */
public class EJBHomeFactory {
  private Map ejbHomes;
  private static EJBHomeFactory aFactorySingleton;

  Context ctx;

  private EJBHomeFactory() throws NamingException {
    //Initialize context with System property Info
    ctx = new InitialContext();
    this.ejbHomes = Collections.synchronizedMap(new HashMap());
  }

  /*
   * EJBHomeFactory 의 싱글튼 인트턴스를 리턴한다.
  **/
  public static EJBHomeFactory getFactory() throws NamingException {
    if ( EJBHomeFactory.aFactorySingleton == null ) {
      EJBHomeFactory.aFactorySingleton = new EJBHomeFactory();
    }
    return EJBHomeFactory.aFactorySingleton;
  }
```

```
    /**
     * EJBHome 객체를 lookup 하고 캐시한다.
    **/
    public EJBHome lookUpHome(Class homeClass) throws NamingException {
      EJBHome anEJBHome = (EJBHome) this.ejbHomes.get(homeClass);

      if(anEJBHome == null) {
        anEJBHome = (EJBHome) javax.rmi.PortableRemoteObject.narrow (
            ctx.lookup (homeClass.getName()), homeClass);
        this.ejbHomes.put(homeClass, anEJBHome);
       }
       return anEJBHome;
    }
  }//EJBHomeFactory
```