JavaOne℠

Enterprise JavaBeans™ (EJB™) 3.1 Technology Overview

Kenneth Saks
Senior Staff Engineer
SUN Microsystems

Agenda

- Introduction
- Proposed Functionality
- Q & A

EJB 3.1 Specification

- Goals
  - Continued focus on ease-of-use
  - New features
- JSR (Java Specification Request) 318
  - Expert Group Formed – August 2007
  - Public Draft – October 2008
  - Proposed Final Draft – March 2009
  - Final Specification planned for Q3 2009

Ease of Use Improvements

- Optional Local Business Interfaces
- Simplified Packaging
- EJB 3.1 “Lite” API
- Portable JNDI Names
- Simple Component Testing

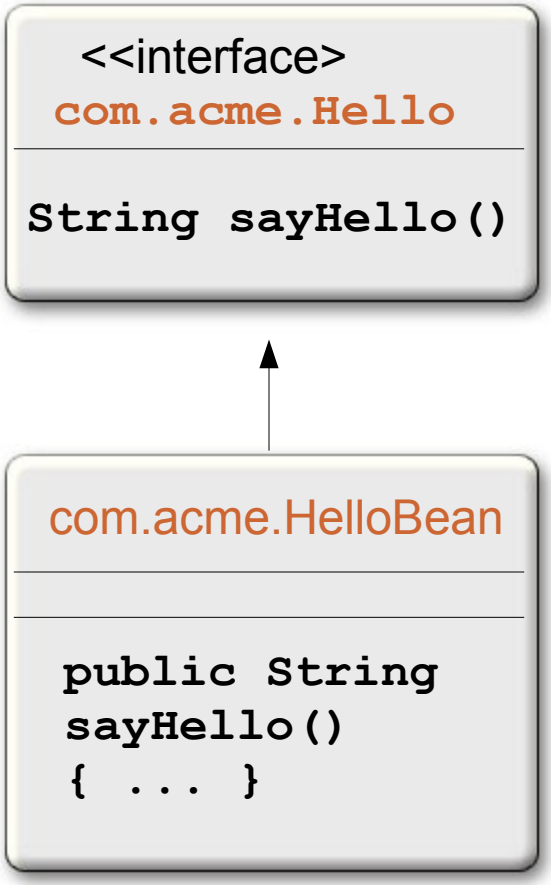
Session Bean with Local Business Interface

HelloBean Client

```
@EJB
private Hello h;

...

h.sayHello();
```



Optional Local Business Interfaces

- Sometimes separate local interface isn't needed
- Better to completely remove interface from developer's view than to generate it
- Result : “no-interface” Local view
  - Just a bean class
  - All public bean class methods exposed to client
  - Client programming model almost identical to local business interface
    - Client does ***not*** call `new()` on bean class

Session Bean with “No-interface” View

```
@Stateless
public class HelloBean {

    public String sayHello(String msg) {
        return “Hello “ + msg;
    }

}
```

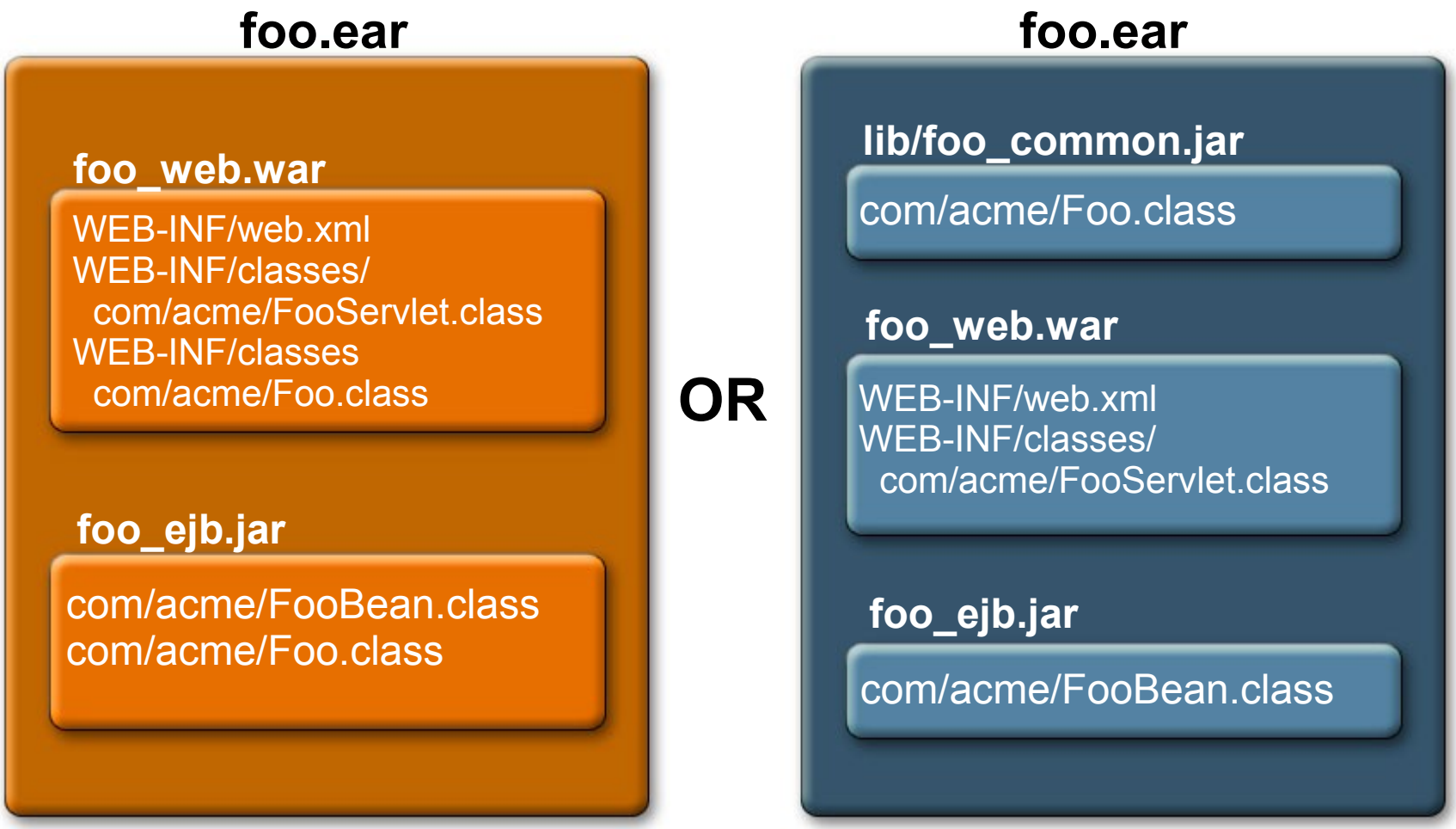
No-interface View Client

```
@EJB HelloBean h;

...

h.sayHello("bob");
```

Java™ EE Platform 5 Packaging



Simplified Packaging

foo.war

```
WEB-INF/classes/com/acme/
  FooServlet.class
  FooBean.class
```

Simplified Packaging

- Goal is to remove artificial packaging restriction
  - Same behavior , without need for ejb-jar
  - Web application APIs are not exposed to EJB components
- Same packaging rules as other .war classes
  - `WEB-INF/classes` or `WEB-INF/lib` .jars
- `ejb-jar.xml` is still optional
  - If needed, placed in `WEB-INF`
- All resources are shared within .war

EJB 3.1 **“Lite”** API

- **Small** subset of EJB 3.1 API required by Java EE Platform 6 Web Profile
- Broadens the availability of EJB technology
  - Without losing portability
- Any 3.1 Lite application can be deployed to Web Profile and Full Java EE 6 Platform
  - Made possible by simplified .war packaging

“Lite” vs. Full

Lite

- Local Session Beans
- CMT / BMT
- Declarative Security
- Interceptors

** Web Profile also includes Java Persistence API

Full = Lite + :

- Message-Driven Beans
- Web Service Endpoints
- 2.x / 3.x Remote view
- RMI-IIOP Interoperability
- Timer Service
- Async method calls
- 2.x Local view
- CMP / BMP Entity Beans

Portable Global JNDI Names

- Spec-defined lookup names for Remote **and** Local session beans
- Three main goals :
  - Simplify intra-application Local Session lookups
  - Simplify mapping of cross-application Remote session bean dependencies
  - Improve portability of Remote Java clients

Portable Naming Syntax

Each session bean gets the following entries :

- Globally unique name

```
java:global[/<app-name>]/<module-name>/<ejb-name>
```

- Unique name within same application

```
java:app[/<module-name>]/<ejb-name>
```

- Unique name within defining module

```
java:module/<ejb-name>
```

No-interface Session Bean

```
@Stateless
public class HelloBean {
   public String sayHello(String msg) {
      return "Hello " + msg;
   }
}
```

If deployed as hello.jar, JNDI entries are:

java:global/hello/HelloBean
java:app/HelloBean
java:module/HelloBean

Intra-module session bean lookup

```
InitialContext ic = new InitialContext();

HelloBean hello = (HelloBean)
  ic.lookup("java:module/HelloBean");

hello.sayHello("bob");
```

Remote Session Bean

```
@Stateless
public class HelloBean implements Hello {

  public String sayHello(String msg) {
    return "Hello " + msg;
  }

}
```

Remote Client

```
// Global session bean lookup

InitialContext ic = new InitialContext();

Hello hello = (Hello)
  ic.lookup("java:global/hello/HelloBean");

hello.sayHello("bob");
```

EJB Component Testing

- It's too hard to test EJB components, especially Local session beans
  - Forced to go through Remote facade or Web tier
  - Separate processes needed for server and client
- Some support for client-side EJB component execution exists, but...
  - Not present in all implementations
  - No standard behavior for bootstrapping, component discovery, shutdown etc.

Example: Local Stateless Session Bean

```
@Stateless
@Local(Bank.class)
public class BankBean implements Bank {

  @PersistenceContext EntityManager accountDB;

  public String createAccount(AccountDetails d) {
    ...
  }

  public void removeAccount(String accountID) {
    ...
  }

}
```

Example: Test Client Execution

```
% java -classpath bankClient.jar :
                  bank.jar :
                  javaee.jar :
                  <vendor_rt>.jar

                  com.acme.BankTester
```

Example: Embeddable API

```
public class BankTester {

  public static void main(String[] args) {

    EJBContainer container =
       EJBContainer.createEJBContainer();

    // Acquire Local EJB reference
    Bank bank = (Bank) container.getContext().
      lookup("java:global/bank/BankBean");

    testAccountCreation(bank);
    ...
  }
}
```

Embeddable API

- Portable API for running EJB components in same process as client code
- Same component behavior / life cycle as server-side
  - CMT/BMT, injection, threading guarantees, etc.
- “Single Application” model
- Only required to support 3.1 “Lite” API
  - Vendors can optionally support additional functionality

New Features

- Singletons
- Startup / Shutdown callbacks
- Calendar-based timers / Automatic timer creation
- Asynchronous session bean invocations
- JAX-RS Integration

Singletons

- New session bean component type
  - Provides easy sharing of state within application
  - Designed for concurrent access
  - One bean instance per singleton component per process
- Lots in common with stateless / stateful beans
  - Client views (Local, Remote, Web Service)
  - CMT / BMT
  - Container services: resource managers, timer service, method authorization, etc.

Simple Singleton

```
@Singleton
public class SharedBean {

  private SharedData shared;

  @PostConstruct
  private void init() {
    shared = ...;
  }

  public int getXYZ() {
    return shared.xyz;
  }

}
```

Singleton Client

```
@Stateless
public class FooBean {

  // Inject reference to Singleton bean
  @EJB
  private SharedBean shared;

  public void foo() {
    int xyz = shared.getXYZ();
    ...
  }

}
```

Singleton Concurrency Options

- Single threaded (default)
  - For consistency with all existing bean types
- Container Managed Concurrency
  - Control access via method-level locking metadata
- Bean Managed Concurrency
  - All concurrent invocations have access to bean instance

Container Managed Concurrency

- Concurrent access determined by method-level locking metadata
  - **READ** lock : allow any number of concurrent accesses to bean instance
  - **WRITE** lock (default) : ensure single-threaded bean instance access
- Incoming invocations are blocked by container until the necessary lock can be acquired
  - … or until an optional timeout has been reached

Read-Only Singleton with Container Managed Concurrency

```
@Singleton
@Lock(READ)
public class SharedBean {

  private SharedData shared;

  public int getXYZ() {
    return shared.xyz;
  }

  @PostConstruct void init() {
    shared = ...;
  }
}
```

Read-Mostly Singleton with Container Managed Concurrency

```
@Singleton
@Lock(READ)
public class SharedBean {

  private SharedData shared;

  public int getXYZ() {
    return shared.xyz;
  }

  @Lock(WRITE)
  public void update(...) {
    // update shared data
    ...
  }
```

Concurrent Access Timeouts

```
@Singleton
public class SharedBean {

  private SharedData shared;

  @Lock(READ)
  @AccessTimeout(value=1, unit=SECONDS)
  public int getXYZ() {
    return shared.xyz;
  }

  public void update(...) {
    // update shared data
    ...
  }
```

Bean Managed Concurrency

- All incoming invocations allowed to access bean instance concurrently
- Bean developer is solely responsible for maintaining integrity of instance state
  - Can use synchronization primitives such as `synchronized`, `volatile`, etc.

Bean Managed Concurrency

```
@Singleton
@ConcurrencyManagement(BEAN)
public class SharedBean {

  private SharedData shared;

  synchronized public int getXYZ() {
    return shared.xyz;
  }

  synchronized public void update(...) {
    // update shared data
    ...
  }
```

App Startup / Shutdown Callbacks

- Uses singleton bean life cycle
- Full container services available
  - Container-managed transactions
  - Entity Managers, Timer Service, etc.
- Startup callback
  - `@Startup` + `@PostConstruct`
- Shutdown callback
  - `@PreDestroy`

Startup / Shutdown Callbacks

```
@Singleton
@Startup
public class StartupBean {

  @PostConstruct
  private void onStartup() {
    ...
  }

  @PreDestroy
  private void onShutdown() {
    ...
  }
}
```

Timer Service Features

- Calendar-based timeouts
- Automatic timer creation
- Non-persistent Timers

Calendar Based Timeouts

- Cron-like semantics with improved syntax
- Can be created programmatically or automatically
- Named attributes
  - `second`, `minute`, `hour`
    - (default = "`0`")
  - `dayOfMonth`, `month`, `dayOfWeek`, `year`
    - (default = "*")
- Relative to optionally specified time zone

Attribute Syntax

- Single value : `minute = "30"`
- List : `month = "Jan, Jul, Dec"`
- Range : `dayOfWeek = "Mon-Fri"`
- Wild card : `hour = "*"`
- Increment : `minute = "*/10"`

Calendar Expression Examples

- "The last Thursday in November at 2 p.m."
  - `(hour="14", dayOfMonth="Last Thu", month="Nov")`
- "Every day at 3:15 a.m. U.S. Eastern Time"
  - `(minute="15", hour="3", timezone="America/New_York")`
- "Every twenty seconds"
  - `(second="*/20", minute="*", hour="*")`

Automatic Timer Creation

- Container creates timer automatically as a result of application deployment
- Logically equivalent to one invocation of `TimerService.createCalendarTimer()` per automatic timer
- Specified via annotation or `ejb-jar.xml`

Automatic Timer Creation

```
@Stateless
public class BankBean {

  @PersistenceContext EntityManager accountDB;
  @Resource javax.mail.Session mailSession;

  // Callback the last day of each month at 8 a.m.

  @Schedule(hour="8", dayOfMonth="Last")
  void sendMonthlyBankStatements() {
    ...
  }

}
```

Non-persistent Timers

```
@Singleton
public class CacheBean {

  private Cache cache;

  @Schedule(minute="*/5", hour="*", persistent=false)
  @Lock(WRITE)
  private void refresh() {
    ...
  }

  @Lock(READ)
  public String getXYZ() {
    ...
  }
}
```

Asynchronous Session Bean Invocations

- *Simple* way to add Remote or Local asynchrony to enterprise bean applications
- "Fire and Forget" or async results via `Future<V>`
- Best effort delivery – persistent delivery guarantees are not required by spec
- Available for Stateful, Stateless, Singleton beans

Local Concurrent Computation Example

```
@Stateless public class DocBean {

  @Resource SessionContext ctx;
  @PersistenceContext EntityManager resultsDB;
  @EJB DocBean myself;

  public void processDocument(Document document) {
    myself.doAnalysisA(document);
    myself.doAnalysisB(document);
  }

  @Asynchronous public void doAnalysisA(Document d) {...}

  @Asynchronous public void doAnalysisB(Document d) {...}

}
```

Asynchronous Results

- Based on `java.util.concurrent.Future`
- Result value is returned via `Future.get()`
  - Also supports `Future.get(long, TimeUnit)`
- Client exception wrapped by `ExecutionException`
  - `getCause()` returns same exception as would have been thrown by a synchronous invocation

Asynchronous Results -- Client

```
@EJB Processor processor;

Task task = new Task(...);

Future<int> computeTask = processor.compute(task);

...

int result = computeTask.get();
```

Asynchronous Results

```
@Stateless
public class ProcessorBean implements Processor {

  @PersistenceContext EntityManager db;

  @Asynchronous
  public Future<int> compute(Task t) {

    // perform computation
    int result = ...;

    return new javax.ejb.AsyncResult<int>(result);
  }

}
```

Async Behavior

- Transactions
  - No transaction propagation from caller to callee
- Method authorization
  - Works the same as for synchronous invocations
- Exceptions
  - Exception thrown from target invocation available via Future<V>.get()

Session Bean as JAX-RS Resource

```
@Stateless
@Path("ssr")
public class StatelessSessionResource {

  @Context
  private UriInfo ui;

  @GET
  public String get() {
    return "GET: " + ui.getRequestUri().toASCIIString();
  }

}
```

For More Information

- Early Access implementation : GlassFish v3
  - http://glassfish.dev.java.net
- Embeddable API BOF (3980)
  - Tuesday, 7:30 p.m.
- JCP : http://jcp.org/en/jsr/detail?id=318
- Blog : http://blogs.sun.com/kensaks/

Agenda

- Introduction
- Proposed Functionality
- Q & A

Thank You

Kenneth Saks
SUN Microsystems
kenneth.saks@sun.com