Business-Driven Software-Engineering
Enterprise JavaBeans I: Introduction & Session Beans

Thomas Gschwind <thg at zurich ibm com>
Agenda

- Introduction
  - From Monolithic Software Systems to Component-Based Software Systems
  - Middleware Systems
  - Component Architectures
  - Service Oriented Architectures

- Enterprise JavaBeans
  - Basics
  - Advantages and Disadvantages
  - The EJB Ecosystem
  - Architecture
  - Session Beans

- Session Bean Demo
Agenda

- Introduction
  - From Monolithic Software Systems to Component-Based Software Systems
  - Middleware Systems
  - Component Architectures
  - Service Oriented Architectures

- Enterprise JavaBeans
  - Basics
  - Advantages and Disadvantages
  - The EJB Ecosystem
  - Architecture
  - Session Beans

- Session Bean Demo
Overview

- Server side component architecture
- Relieves programmer of writing complex component framework
- Simple construction of enterprise class distributed applications
- Portability & reusability (across EJB servers due to standardized interfaces)
- Rapid application development

Enterprise JavaBeans provides a server-side component architecture. A server-side component architecture consists of a set of components running on a backend-application server that provides common services such as security or transaction management to the components. Developers writing Enterprise JavaBeans are relieved from writing a complex component framework providing these common services and thus simplifies the construction of large enterprise class distributed applications. Due to its reliance on standards, EJB enables the sharing of software components among multiple applications and therefore speeds up the application development.

So what is a software component?
Software Components

- A component is an abstract part of something
- A software component is a piece of code written to manifest the behavior of a corresponding abstract concept
- Generally, components differ from objects
  - More coarse grained granularity
  - Longer lifetime
  - Can stand on their own and thus be reused more easily
  - Can be reused without deep software engineering knowledge (e.g., from business processes)
- Paved the way to component architectures (e.g., CORBA)

Before we proceed we need to answer the question, what a software component is. According to WordNet, a component is an abstract part of something. With the term software component, however, we go one step further and define it as a piece of code written to manifest the behavior of a corresponding abstract concept such as a Mortgage component which may encapsulate all necessary information to handle a mortgage. While this may sound similar to objects, components differ in the following characteristics:

- Generally, they are more coarse grained
- Software components typically have a longer lifetime
- They can stand on their own and can be reused more easily in other applications

Software components pave the way to component frameworks such as the Common Object Request Broker Architecture (CORBA) and other similar component architectures.

What are the challenges in moving from a historical system to such a component architecture?
Historically, we have a monolithic system where the entire application runs on a single mainframe computer. As long as it is possible to run such a system on a mainframe, everything is fine but if the system needs to be distributed and broken up into several pieces, many challenges arise. Which challenges can you think of?
Challenges

- Remote Method Invocation
- Load balancing
- Transparent fail-over
- Back-end integration
- Transaction management
- Clustering & replication
- Update management (redeployment)
- Logging & auditing

- Systems management
- Threading
- Message-oriented middleware
- Object life cycle
- Resource pooling
- Security
- Caching
- ...

In order to solve these challenges middleware systems have been developed. Originally, these systems were very simple and only supported the interaction between distributed objects.
Application Servers

- Provide all these middleware services
  - Requires less in-house expertise
  - Developers can focus on the requirements of the application
  - Application can be assembled more quickly
- Higher quality of enterprise applications
  - Used more frequently => bugs detected earlier
- Lower cost of ownership
  - Components may be replaced more easily
  - Component vendors under pressure
  - Need to provide better documentation

In order to solve the above challenges, application servers have been developed. Application servers provide all these middleware services as built-in feature and hence require less in-house expertise. For instance, developers do not need to know how to implement a transaction manager. Instead, they can focus on the requirements of the application and subsequently because the middleware services are provided by the application server in a consistent manner allows applications to be assembled more quickly.

Other benefits of an application server are that it will be used more frequently and bugs identified in the application server will also benefit all other users of the same application server, resulting in a better quality of the applications themselves.

At the same, the cost of ownership can be lowered, because components can be replaced more easily, which in turn puts more pressure on component vendors to provide components of higher quality and with better documentation. If one component vendor does not provide components of adequate standard, a customer can switch provider more easily. This is in stark contrast to old monolithic systems where customers mostly have been locked in to a single vendor due to a lack of interoperability.
Effectively these systems solve the remote method invocation and the location of other objects on the network. Traditionally, distributed systems communicated over the network via the use of network stubs that were responsible to serialize the data of the client request such that it can be sent over the network, deserialized on the server and passed to the distributed object and the same again in reverse for the object’s response.
... via Explicit Middleware ...

```java
boolean buy(Stock s, Quantity q, Price p) {
    // 1. authorize request
    // 2. start transaction
    // 3. retrieve data from database
    // 4. perform request
    // 5. store data in database
    // 6. commit transaction
}
```

- Steps 3-5 frequently can be combined
- For long requests, overhead is smaller
- Security system cannot be changed
- Transaction processing cannot be changed

The next step after distributed systems were explicit middleware. Explicit middleware requires server-side components to explicitly interact with the middleware system. The above figure shows the steps necessary in most server side components. First, the component has to identify whether the client is allowed to issue the request, then it needs to start a transaction or register with an already existing transaction. Next, it needs to retrieve the data it wishes to work on from the database, manipulate the data, and write it back to the database. Once all of these steps succeed, the transaction has to be committed (unless the request is performed as part of a bigger transaction).

This can pose a substantial overhead on developers since the above steps are typically similar from server-side component to server-side component. Another, more serious drawback, is that with such code the authentication and transaction processing code can only be changed within the code itself.
The above graphic shows how the individual parties communicate with each other when using explicit middleware systems.
The next step, beyond explicit middleware, is implicit middleware. Using explicit middleware, the server-side components only need to know about how to perform the request but not necessarily about authentication, authorization, or transaction management. It simply is sufficient to write the code for the request. While this relieves software engineers, about writing the code to interact with the middleware system, the developer still has to know about transactions and which users should be able to interact with the service, which is information that has to be put into a configuration file. This configuration file can be changed easily at a later time without having to touch the code.
The above graphic shows how the individual parties communicate with each other when using implicit middleware systems. One potential disadvantage with this setup is that it separates security parameters from the code it deals with and puts it into a separate configuration file. This problem has also been identified in EJB. Hence, EJB allows developers to specify default parameters in the source code and subsequently allows these defaults, if necessary to be overridden with configuration files.
Application Servers (cont’d)

- Many application servers emerged in the past
- Each implemented own proprietary interface
- No agreement on a standard
  - Customers locked in to specific vendor
  - No commerce of components
  - Little advantages => no (small) component marketplace

In the past, many application servers have emerged but each implemented its own proprietary interface. As a result there was no agreement on how to interact with middleware services and customers were locked in with a given application server provider. Also the existence of multiple different incompatible application servers fragmented the component marketplaces and from this point of view provided little advantage over the previous state of the art.
Component Architectures

- What is a Component Architecture?
- Agreement on interfaces
  - Application server
  - Components

=> Components may be used on multiple application servers

The lack of interoperability between application servers has been addressed by component architectures such as Enterprise JavaBeans. Component architectures are an agreement on interfaces that both vendors of application servers (implementing that component architecture) as well as software components adhere to. As a consequence, components may be used on multiple application servers.
Component Marketplace

- Has been around for >15 years
- Problems
  - Maturity
  - Politics
    (only a locked in customer is a loyal customer)
  - Questionable value
    (not customer-centric)

Will a component marketplace emerge?

The term “Component Marketplace” has been around for over 15 years and for a long time it has plagued with the problems of a lack of maturity, which partly is due to politics of some vendors who believe that only a locked-in customer will be a loyal customer, or because some clients believe that it is of questionable value to them because off the shelf components are not tailored to their particular need.

This poses the obvious question that given these problems will there ever be a component marketplace to emerge? Today a component marketplace is already available although in a slightly different form. Instead of selling individual components, companies are selling them together with the service to customize them for the customer as part of outsourcing deals.
Service Oriented Architecture

- Services are at the core of SOA
  - Group of components carrying out a business task
  - Provide higher level of abstraction from functional point of view
  - Autonomous, loosely coupled
  - Interact with each other independent of implementation technology or platform

- Complements Component Architecture

Another term frequently used in combination with application servers and component architectures are service oriented architectures. As the name suggests, at the core of service oriented architectures, we can find services. Services are implemented by a group of components carrying out a business task. Services provide yet a higher level of abstraction and support autonomous, loosely coupled services. One of the advantages of such services is that they facilitate the communication across different systems and interacting with each other independently of the implementation technology (e.g., Java, C#).

Service oriented architectures complement component architectures.
Web Services

- One approach for implementing Service Oriented Architectures
- Group of XML Technologies
  - SOAP: Protocol Independent Standard for Exchanging Messages
  - WSDL: Used to describe the Web Service Interface

One approach to implement service oriented architectures are web services. Web services are a group of XML technologies such as SOAP, the protocol that defines the message exchange format, or WSDL which is used to describe the interfaces of web services.
Agenda

- Introduction
  - From Monolithic Software Systems to Component-Based Software Systems
  - Middleware Systems
  - Component Architectures
  - Service Oriented Architectures

- Enterprise JavaBeans
  - Basics
  - Advantages and Disadvantages
  - The EJB Ecosystem
  - Architecture
  - Session Beans

- Session Bean Demo
Enterprise JavaBeans (EJBs)

- Not to be confused with JavaBeans
  - Java Beans => (mostly) GUI components
  - EJBs => sit behind the GUI
- Provides a component architecture
- EJB is only a standard
- EJBs accessed from
  - Thick clients
  - Servlets, JSP, etc.
  - Web service wrappers

Enterprise JavaBeans is Sun’s answer to the requirements by today’s component architectures and service oriented architectures. Enterprise JavaBeans, however, should not be confused with JavaBeans which purely reside on the desktop and typically provide GUI components, although the same naming scheme as for JavaBeans should be used for Enterprise JavaBeans.

Enterprise JavaBeans provide a component architecture that supports the separation between the individual components and the application server. That is, components and application servers can be easily mixed among each other. This is accomplished by Enterprise JavaBeans just being a standard that has to be implemented by EJB application server providers.

Enterprise JavaBeans can be accessed from a wide range of different clients, such as thick clients, servlets or JSPs, and web service wrappers.
Advantages of EJB

- Rapid application development
- Well-defined standard
  - Standard is freely available
  - Portability
    (Test Suite, Reference Implementation)
- Widespread use
  - Easier to find qualified employees
  - Easier to find an employer
- Integrates nicely with the JEE Platform

- Certainly, nothing is perfect and we will see where EJBs could be improved as well

The advantages of EJB are that it defines the interfaces of an application server and thus provides for rapid application development by relieving developers to implement low level middleware services. The standard is freely available and a test suite and a reference implementation are available to allow vendors to verify whether their implementation is correct. Enterprise JavaBeans has been adopted by many vendors (e.g., IBM, BEA) and is in widespread use. This makes it easier for companies to find qualified employees and for software engineers with EJB knowledge easier to find an employer. The EJB specification has been from ground up specified to nicely integrate with the Java EE platform. Later during the lecture we will see that EJB also has some shortcomings where the specification could be improved.
As the picture demonstrates, the J2EE platform supports the development of rather complex applications. In order to manage all these tasks in such a system, the EJB Ecosystem divides the tasks in such an environment into multiple different roles that may be executed by different people.
The different parties involved in the EJB ecosystem are:

- the tool provider that creates the tools for assembling and developing EJB applications (these tools are used by all other parties except for possibly the EJB container/server provider),
- the EJB container/server provider that provides the application server where the EJB components are deployed on,
- the bean provider developing the beans,
- the application assembler that takes multiple beans and uses them to create the EJB application,
- the deployer who deploys an application on the application server, and
- the system administrator who is administering the running system.

In many production environments, however, many of these roles will be realized by the same person, especially, bean provider and application assembler, and deployer and system administrator.
EJB and Java

- Tied to the Java programming language
- Uses standard Java APIs for
  - Performing business logic (e.g., JavaMail API)
  - Accessing databases (Java Database Connectivity (JDBC) API)
  - Integration with existing applications (Java Connector Architecture (JCA))

Enterprise JavaBeans is tied to the Java programming language and heavily relies on standard Java APIs for performing the business logic, accessing databases, or the integration with existing applications.
The above table shows the Java APIs that EJB builds upon.
E-Commerce Application

- Customers select goods
- Application server updates a shopping basket accordingly
- Customer confirms order
  - Check customers credit
  - Update inventory
  - Trigger shipment of an order
  - Bill customer

A sample EJB application could be an e-commerce application, where the customer selects a set of goods, and the application updates the shopping basket. At the end, the customer confirms the order, and the EJB application checks the customer’s credit worthiness, updates the stock inventory, triggers the shipment of the order and bills the customer. This application makes it clear that EJB is most beneficial for larger applications where a large fraction of the underlying business is to be automated. For instance, for a small company where only the shopping basket has to be implemented electronically, and the rest is performed by hand, EJB might be an overkill (unless the application is to be expected to grow).
Enterprise Beans

- Server-side component
- May be composed of multiple Java objects
- Exposes single interface to clients
- Deployed in distributed multi-tier environment
- May implement life-cycle methods (e.g., for construction & destruction)

The individual components in EJB are typically referred to simply as “Enterprise Beans.” They are server-side components that run on a server and are frequently composed of multiple Java classes. Despite being composed of multiple Java classes, they only expose a single interface to their clients.

Enterprise beans are deployed in a multi-tier environment where clients, application logic, and database logic are running on separate tiers. Additionally, enterprise beans may implement a set of life-cycle methods to allow them to respond for instance to their creation and destruction.
The following types of beans exist:

- Session beans that typically represent a process, task or workflow such as a shopping website, a stock trading application, or library service. Session beans talk to other beans and are the “only” beans responsible for the communication between different beans.

- Entity beans and JPA entities represent data and information such as an account or a portfolio item. They must provide a primary key to allow them to be distinguished from one another. Entity beans are mostly superseded by JPA entities.

- Message-Driven Beans are similar to session beans but are invoked by clients sending JMS messages (i.e., they act as message sinks).

Theoretically, beans can be misused for any purpose but this would make the enterprise application hard to understand. In the past this has been done to get around certain limitations of earlier versions of the EJB specification but is no longer necessary.
The above picture depicts a simple EJB component system showing how the different beans could potentially interact with each other.
Constituents of an Enterprise Bean

- Implementation Class
- Business methods
  - Remote Interface
  - Local Interface
- Deployment Descriptor
- EJB Object (container generated)

- Pre-EJB 3.0
  - Home Interface
  - Local Home Interface
  - Home Object (container generated)

Enterprise beans consist of the following constituents:

- The implementation class that implements the bean’s business logic and business methods
- The business methods which are exposed in the remote and/or local interface
- The deployment descriptor that specifies the configuration of the middleware services (i.e., security attributes, transaction boundaries, etc)
- The EJB object which is automatically generated by the EJB server during deployment and mostly invisible for the software engineer

Previously to EJB 3.0, an enterprise bean also consisted of the home interface and/or local home interface and a container generated home object. In the pre EJB3.0 specifications, these were responsible for dealing with the bean’s lifecycle. In the EJB 3.0 era they are no longer necessary. Pre EJB 3.0 beans will be discussed in another lecture.
Since EJB 3.0, the implementation is provided as part of a Plain Old Java Object (POJO). The class, however is annotated with Java annotations. The annotations used in the above class are:

- `@Stateless` indicates that this is a stateless session bean. The type of session bean is a `@Stateful` session bean.
- The `@Remote` annotation indicates the remote interface. This annotation, however, is optional if the bean implements the remote interface and the remote interface has been annotated with `@Remote`
- Similar to the `@Remote` annotation, there is a `@Local` annotation

Any of the annotations, however, may be overridden within the deployment descriptor.
The Remote Interface

- Each EJB Object exposes a remote interface
- Lists methods provided to remote clients
  - These methods will be wrapped by the EJB Object

```java
@Remote
public interface QuoteServiceRemote {
    public String getQuote();
}
```

The remote interfaces exposes those methods that should be available remotely. The remote interface is annotated with the @Remote annotation.
In order to invoke a method, the call goes through the client-stub, responsible for marshaling the parameters, the data is sent over the network to the server, where the server skeleton unmarshals the parameters, extracts the method to be invoked, passed the information to the request interceptor, which finally invokes the EJB method. In order to pass the return value back to the client the whole process has to be repeated again in reverse order.

If we wanted to invoke a bean residing within the same virtual machine, performing all this would be inefficient. However, invoking the bean directly is not possible since then the EJB container would not be able to take care of the security and transaction management across bean boundaries. This is why the EJB specification provides local interfaces.
The Local Interface

- Remote interfaces are slow for local clients (marshaling)
- Solution: local interfaces
  - If bean to be called by another local one
  - Simple wrapper for security & transaction management
  - Passes arguments directly (i.e., pass by reference!)

- Ouch! Did anybody say location transparency?
  Or access transparency?

Remote interfaces are slow for the interaction with local clients. This is because all parameters are marshaled/serialized when the remote interface is being used. The solution of the EJB developers is to provide local interfaces. If a bean’s method is invoked via the local interface, the bean’s method is invoked without marshaling/serializing the parameters.

Essentially, the local interface provides a simple wrapper around the bean to handle the beans security and transaction management requirements.

This architecture, however, violates two of the principles of distributed computing: location and access transparency. Location transparency, because the client has to know whether the bean is locally available or not. Access transparency because local and remote interfaces are accessed differently due to different parameter passing semantics and possibly different methods exposed in the two interfaces.
The above figure shows how an enterprise bean is invoked through a local interface. The marshaling can be skipped and the request interceptor is invoked directly. This also shows why an enterprise bean must never be invoked directly (i.e., without going through the EJB infrastructure), because otherwise the request interceptor (i.e., EJB object) would not be able to start/stop any transactions or interact with any of the other middleware services necessary for the bean’s operation.
Remote vs. Local Interface (cont’d)

- **Remote interface**
  - Accessible to remote clients but slow for local clients
  - Arguments passed by value
    - (bean operates on its own copy of the values)

- **Local interface**
  - Fast but only accessible to local clients
  - Arguments passed by reference
    - (bean and client operate on same copy of objects)

- Remote and local interface may expose different methods

- Security, Transaction Management, etc.
  - No difference (both invoked through an EJB object)

---

The above slide again summarizes the differences between local and remote interfaces. Another caveat is the different call semantics.

Invocations via the remote interface use a pass by copy semantics. That is all data structures are being copied and the remote bean modifies its own local data structure and modifications are not visible to the client.

Invocations via the local interface use pass by reference semantics. That is all data structures are passed by reference and the client and bean operate on the same copy of the data structure and modifications done by the bean will be visible to the client.
The Deployment Descriptor

XML file that specifies
- Bean management
- Lifecycle requirements
- Security requirements
- Transaction requirements
- Persistence requirements (Entity Beans)

The deployment descriptor is an XML file that specifies the bean’s general information such as name, JNDI name, etc., the bean’s life-cycle requirements, security and transaction attributes, and in case of entity beans (not JPA entities) the persistence requirements.
The EJB Object

- Clients do not interact with the EJB directly
- Surrogate object that knows about networking, transactions, etc.

Advantages
- Security & Transaction management
- Resource & life cycle management
- Threading support
- Monitoring
- Implicit remote accessibility
- Location transparency

The EJB object is responsible for taking care of the bean’s security and transaction management. Clients do not interact with the bean directly but through the EJB object. This is a surrogate object that knows how the bean wishes to interact with the middleware services and does this for the bean implementation. This provides the previously discussed advantages that the bean no longer needs to take care of middleware services such as security or transaction management, resource and life-cycle management, etc.
The EJB Object (cont’d)

The above figure depicts the steps that occur when a client invokes a method on an enterprise bean.
Session Beans

- Reusable components representing business logic
- Represent process/task/workflow (e.g., shopping or stock trading application)
- Establish link between beans
- Session beans may be stateless or stateful

Session beans represent the business logic of the enterprise application. That is, the process, task or workflow such as a shopping website, a stock trading application, or library service. Session beans talk to other beans and are the “only” beans responsible for the communication between different beans. Session beans can either be stateless (i.e., storing no communication state) or stateful (i.e., storing communication state).
Session Bean’s Life-cycle

- Created by EJB container when needed by a client
- Session beans are short-lived
- Roughly the lifetime of a session or of the client code
- Lifetime typically determined by the length of the client’s session

Session beans are created by the EJB container when they are needed by a client. Typically, session beans are short-lived meaning that they live as long as the session between the client and the EJB server is active whereas entities may live longer than an individual session. Hence, a session bean’s life-cycle is roughly determined by the client’s session.
Stateless Session Beans

- For business processes that comprise a single request
  - Perform some computation
  - Request trivial information (e.g., exchange rates)
- No state is stored on behalf of the client
- Client has to pass all required information with each request
- Bean may be destroyed or recycled after each request
  - Be prepared when implementing such beans

Stateless session beans do not store any communicational state. That is they do not keep information about previous requests. This makes them suitable for application logic that can be computed within a single request or where it is trivial for the client to pass in all parameters necessary for processing the request over and over again. A sample such service could be to request exchange rates. The advantage of this type of bean is that the bean may be reused for any request from any client at any time.
The above figure shows that the EJB container may use any stateless session bean at any time (unless the bean services already a request for another client – there are never two different threads accessing the same bean simultaneously) to service a client request.
The methods invoked during the beans life-cycle.
- `ejbCreate()` refers to any method marked with `@PostConstruct`
- `ejbRemove` refers to any method marked with `@Remove`
Stateful Session Beans

- For business processes that last for several requests
  - Adding, removing products to a shopping cart
  - Banking process where multiple transactions are being performed
- Stateful session beans retain state of an individual client

Stateful session beans store information regarding the communication state. For instance, which client is invoking the bean. This may be useful for an application where a client is expected to invoke multiple methods whose execution is related to each other and where it would be difficult for the client to pass all state information to the bean over and over again.

An example allowing both the use of a stateless or stateful bean would be a bank application.

- In case of the stateful bean, the client would log in and subsequently can perform operations related to his accounts.
- In case of a stateless session bean, the id of the customer interacting with the bean would have to be passed with every request.
Stateful Session Bean Pooling

- Cannot be pooled as easily
- Session state needs to be preserved
  - May be swapped out (bean is passivated)
  - And swapped in (bean is activate)
- Callbacks for bean specific cleanup and initialization
  - @PrePassivate annotated methods
    (or ejbPassivate() in pre EJB3.0)
  - @PostActivate annotated methods
    (or ejbActivate() in pre EJB3.0)

Stateful beans are being pooled and reused for requests from different clients. However, this is more complicated for stateful beans than for stateless beans where any bean can be used.

In case of stateful beans, if all bean instances are already in use for other sessions, the EJB container has two options:

- Create a new bean (which is possible unless the maximum number of beans to instantiate has been reached)
- Reuse a bean that is servicing another client’s session. In this case the other client’s session has to be passivated and bean loaded with our communication state. If the old client comes back another bean may have to be passivated and the old client’s state activated again.
Stateful Session Bean Pooling

Client

invoke()

Remote Interface

EJB Object

Find lru bean
Call ejbPassivate
Serialize bean

EJB Container/Server

Bean

Bean

Bean
Stateful Session Bean Pooling

Client

invoke()

Remote Interface

EJB Object

EJB Container/Server

Deserialize bean
Call ejbActivate
Invoke method

Bean

Bean

Bean

Bean
Stateful Session Bean Pooling

- Typically accomplished using Java serialization
  - ...Bean extends ...Serializable

- Container must ensure that the following are preserved
  - Non-transient variables
  - EJB Objects
  - Home Objects
  - EJB Context References
  - JNDI Naming Contexts
Lifecycle of Stateful Session Beans

Bean Instance Does Not Exist

Class.newInstance()
setSessionContext()
ejbCreate()

Client times out

Business Method

Ready

ejbPassivate()
ejbActivate()

Passive

ejbRemove()
Agenda

- Introduction
  - From Monolithic Software Systems to Component-Based Software Systems
  - Middleware Systems
  - Component Architectures
  - Service Oriented Architectures

- Enterprise JavaBeans
  - Basics
  - Advantages and Disadvantages
  - The EJB Ecosystem
  - Architecture
  - Session Beans

- Session Bean Demo
Summary

- Introduction
  - From Monolithic Software Systems to Component-Based Software Systems
  - Middleware Systems
  - Component Architectures
  - Service Oriented Architectures

- Enterprise JavaBeans
  - Basics
  - Advantages and Disadvantages
  - The EJB Ecosystem
  - Architecture
  - Session Beans

- Session Bean Demo
Review Questions

- What is the difference between implicit and explicit middleware?
- What is a Component Architecture?
- What are the different types of Enterprise JavaBeans?
- What are the constituents of an EJB 3.0 session bean?
- What is the difference between the local and remote interface?
Tasks

1. Download and install
   - Java
   - Eclipse JEE
   - JBoss

2. Download the QuoteService session bean from the BDSE homepage and get it up and running

3. Have a look at the sources and try to understand its constituents
Environment

- Download the Java SDK
- Download Eclipse-JEE-Helios
- Download Jboss 5.1.0 Application Server
  - Jboss 6.0 has been released but we won’t use it
- Unpack and install the above
How to Start

1. Run Eclipse
2. Create a Server project (File/New/Other…/Server/Server)
3. Start the server in the Servers view
4. Create an EJB Project or import the QuoteService project (Import…/General/Existing Projects into Workspace/Select archive file:)
Troubleshooting
Cannot find console output

- There will be multiple consoles
  - One for the server
  - One for your client
  - Potentially one for eclipse’s deployment command

- You can switch between those in the toolbar of eclipse’s console view
java.lang.IllegalArgumentException on JBoss startup

- Edit
  $JBOSS_HOME/server/$servername/conf/bootstrap/profile.xml
- Locate “<bean name="AttachmentStore" ...”
- Replace “<parameter>” tag in next line with “<parameter class="java.io.File">“
Jboss Cannot Bind Name

- Deleted an EJB Project while it was deployed?
- Go to $JBOSS_HOME/server/$config/deploy and remove it manually
Eclipse cannot find javax.ejb.* classes

- Missing EJB Environment in EJB Project
- $project/Properties/Java Build Path/Libraries/Add Library/Server Runtime and add the JBoss server to the project
Outlook

- Session Beans
- Java Persistence API

See you next Tuesday!