



EJB3 in Action
by Debu Panda,
Reza Rahman,
and Derek Lane

Unedited draft
Chapter 2

Copyright 2006 Manning Publications

Chapter 2 *A First taste of EJB*

2.1 The ActionBazaar Application	2
2.2 Building business logic with Session Beans.....	4
2.2.1 Stateless Beans	4
Stateless Bean Client.....	5
2.2.2 Stateful Beans	6
Stateful Bean Client	8
2.3 Messaging with Message Driven Beans	9
2.3.1 Producing a Billing Message	10
2.3.2 The Order Billing Message Processor	12
2.4 Persisting data with EJB 3.0	13
2.4.1 A Persistence API example.....	14
2.5 Summary	17

In this age of hyper-competitiveness, learning a new technology by balancing a book on your lap while hacking away at the business problem on the keyboard has become the norm. But let's face it—somewhere deep down, you probably prefer “baptism by fire.”

This Chapter is for the brave pioneer in all of us, who can take a quick look at all the EJB types and be satisfied with learning the details in subsequent chapters. To stick to the basics, the examples are stripped down to bare bones, exposing only the very high level features of EJB 3.0. If a view from the heights seems frightening, think of this chapter as that first day at a new workplace, shaking hands with the stranger from the next cubicle. In the chapters that follow, you will get to know more about your new co-workers' likes, dislikes, eccentricities and how to work around these foibles. But this chapter is designed to show you exactly how easy and useful EJB 3.0 is.

Before we jump into code, we introduce you to an important element of this book – the *ActionBazaar* application. All of the material in this book is essentially woven around developing parts of this imaginary enterprise system.

2.1 The ActionBazaar Application

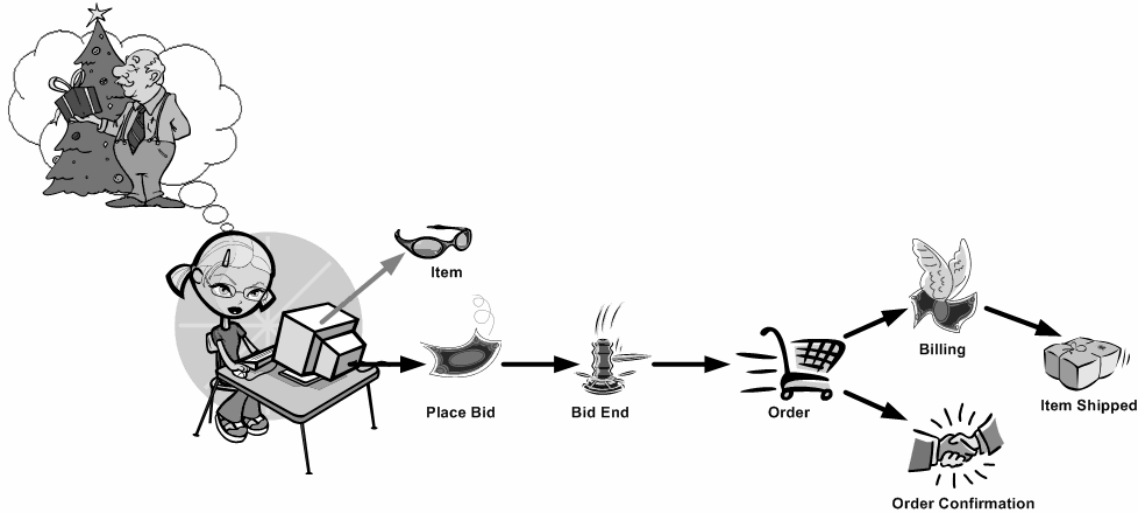
ActionBazaar is a simple online auctioning system like eBay. Sellers dust off their treasures, take blurry pictures, and post their items on ActionBazaar. Buyers get carried away and bid exorbitantly. Winning bidders pay for the items. Sellers ship items. Everyone is happy, or so the story goes.

The second two parts of the book roughly follow the course of developing each layer of this ActionBazaar application as it relates to EJB 3.0. We mentioned in Chapter 1 that a typical enterprise application has three tiers: presentation, business logic and persistence layers. As you know, unlike JSF, JSP, Struts and the like, EJB 3.0 is not a user interface layer technology. Consequently, we will not deal with the presentation layer beyond what is absolutely necessary. Instead, we will use EJB 3.0 to develop the business logic tier in Part 2 of this book, and then the persistence tier in Part 3.

As much as we would like to take credit for it, the idea of ActionBazaar was first introduced in *Hibernate in Action* by Christian Bauer and Gavin King as the *CaveatEmptor* application. The *Hibernate in Action* book primarily dealt with developing the application persistence layer using the Hibernate O-R framework. The idea was later used by *Webworks in Action* to discuss the popular open source web framework. This book follows in these footsteps.

The code samples used throughout this book can be downloaded from <http://ejb3inaction.com>. We highly recommend that you follow the tutorial on the site to set up your development environment so you can follow along with us and experiment on your own as you wish.

For the purposes of quickly introducing all three EJB 3.0 types across the business logic and persistence layers in this Chapter, we will focus on a single representative chain of functions in ActionBazaar—starting from bidding on an item and ending in ordering the item. The business function chain is represented in Figure 2.1.



1 **Figure 2.1: A chain of representative ActionBazaar functionality used as an example to quickly examine a cross-section of EJB 3.0 functionality. The bidder bids on a desired item, wins the item, orders it, and instantaneously receives confirmation. In parallel to order confirmation, the user is billed for the item. Upon successful receipt of payment, the seller sells the item.**

In a sense, the functions represented in Figure 2.1 are the “essentials” of ActionBazaar. In addition, ActionBazaar sells some cheap accessories that go with the items being auctioned. These items are sold only to winning bidders. The only two major functionalities not covered in this chapter are the process of posting an item for sale and searching for items. We will save these two pieces of functionality for the chapters in Parts 2 and 3.

The chain of actions in Figure 2.1 starts with the user deciding to place a bid on an item. In Section 2.1.1 we will see how the action of placing a bid can be implemented with a *Stateless Session Bean*. After the bidding cycle is completed and the timed auction ends, the highest bidder wins the item. The winning bidder then orders the item with some cheap accessories; this process includes specifying billing and shipping information. Later in this chapter, we will see how a *Stateful Session Bean* implementing the ordering process might look.

When studying the workings of the ActionBazaar, note that it is unlike many e-Businesses like Amazon.com and eBay, ActionBazaar processes billing asynchronously. This means that the user does not have to wait for the billing process to finish before receiving an order confirmation. Instead, the order confirmation is generated as soon as the ordering information is entered and reasonably validated. The actual billing process, on the other hand, takes place in parallel, as indicated in Figure 2.1. Later sections of this chapter explain how *Message Driven Beans* are used to asynchronously bill the user for an order.

After the billing process is finished, both the seller and bidder are notified and the seller ships the item. Since you probably have application development experience, it should be obvious to you that at each step of the action chain, a good deal of data is persisted. The persisted data includes the bid, order, billing and confirmation data that is saved into a relational database as needed. To examine how the *EJB 3.0 Java Persistence API* works, we will see how a bid is stored into the database in sections that follow.

Now that you understand the whirlwind tour of EJB 3.0 you’re embarking on, let’s start with *Session Beans*, the enterprise business-processing workhorse.

2.2 Building business logic with Session Beans

We use Session Beans to implement both the bidding and ordering processes. In general, these Beans are meant to encapsulate business logic and model atomic processes or actions, especially as perceived by the system user. In this sense, they can be categorized as the easiest but most important part of EJB. When we examine Session Beans in much greater detail in Chapter 4, you will see that they are designed to provide high performance and concurrency in a virtually transparent manner. In addition, you will see in Chapters 3, 4, and 6 that Session Beans provide a whole host of functionality specifically geared toward implementing robust business logic quickly.

Session Beans come in two flavors – Stateful and Stateless. We will take on Stateless Session Beans first because they are commonly considered to be simpler.

2.2.1 Using Stateless Beans

The ‘Hello World’ example has ruled the world since it first appeared in *The C Programming Language* by Kernighan and Ritchie. ‘Hello World’ caught on and held ground for good reason. It is very well suited to focus on introducing a technology as simply and plainly as possible. As a rowdier bunch, we are going to give you something a little more substantial from the get-go. The first code sample of this book is the `PlaceBid` Stateless Session Bean in `ActionBazaar`. The `addBid` Bean method is called from the web tier client in response to a user request to place a bid on an item. The parameters to the method specify the ID of the bidder placing the bid, the ID of the item being bid on, and the bid amount. The method adds the specified bid to the database. Since the code sample is already long enough, we have omitted as much code as we could, including the definition of the `Bid`, `Bidder` and `Item` objects.

1 Listing 2.1: PlaceBid Stateless Session Bean code

```
package ejb3inaction.example.buslogic;

import javax.ejb.Stateless;                                     | #1
import ejb3inaction.example.persistence.Bid;
import ejb3inaction.example.persistence.Bidder;
import ejb3inaction.example.persistence.Item;

@Stateless(name="PlaceBid")                                    | #2
public class PlaceBidBean implements PlaceBid {                | #3
    public PlaceBidBean() {
    }

    public Bid addBid(String bidderId, Long itemId,             | #4
        Double bidAmount) {                                     | #5
        Bid bid = new Bid();

        Bidder bidder = ... Lookup bidder using bidderId ...   | #6
        bid.setBidder(bidder);                                  | #7
        Item item = ... Lookup item using itemId ...            | #8
        bid.setItem(item);                                      | #9
        bid.setAmount(bidAmount);                               | #10

        ... New bid saved into the database ...

        return bid;                                           | #11
    }
}
...

```

```

package ejb3inaction.example.buslogic;

import javax.ejb.Remote;                                     | #12
import ejb3inaction.example.persistence.Bid;

@Remote                                                     | #13
public interface PlaceBid {                                 | #14
    Bid addBid(String bidderId, Long itemId,               | #15
        Double bidAmount);
}
(annotation) <#1 Import bean type annotation>
(annotation) <#2 Mark as Stateless Session Bean>
(annotation) <#3 Plain Old Java Object Bean Implementation>
(annotation) <#4 Bean business method>
(annotation) <#5 Create new bid>
(annotation) <#6 Lookup bidder>
(annotation) <#7 Set bidder>
(annotation) <#8 Lookup item>
(annotation) <#9 Set item>
(annotation) <#10 Set bid amount>
(annotation) <#11 Return bid saved into database>
(annotation) <#12 Import remote interface>
(annotation) <#13 Mark Bean interface as remote>
(annotation) <#14 Bean interface>
(annotation) <#15 Bean business method definition>

```

The first thing you should note is how “normal” this code looks. This is particularly striking if you have done any work with EJB 2.x. In EJB 3.0, everything is either a Plain Old Java Object (POJO) or Plain Old Java Interface (POJI).

There is no cryptic interface to implement, Class to extend, or puzzling naming convention to follow. At best, all you need to do is use a few metadata annotations like the `@Stateless` and `@Remote` annotations used in Listing 2.1 (recall the discussion on metadata annotations in Chapter 1).

As mentioned, we will not try to dwell on the code now and will save detailed coverage for later chapters. We will, however, note the major features in Listing 2.1. Both the `PlaceBidBean` Bean implementation Class and `PlaceBid` business interface belong in the `ejb3inaction.example.buslogic` package and are stored in separate .java files. The `@Stateless` annotation tells the container that `PlaceBidBean` is a Stateless Session Bean#2. Like any other Java type, annotations must be imported, as in #1 and #12. The `@Stateless` annotation’s name parameter specifies the name of the EJB to be ‘PlaceBid’#2. As we will soon see, the client invokes the `addBid()` business method#4 through the `PlaceBid` interface. The `@Remote` annotation tells the container that the `PlaceBid` EJB can be called by remote clients#13. This means that the client can be running in a different JVM than the EJB, very likely on a separate machine. If you are curious to see the rest of the code, you can download it from <http://ejb3inaction.com>. The `chapter2.zip` file contains the source code and build scripts needed to deploy the application. Let us now turn our attention to the client code for using the `PlaceBid` EJB.

The Stateless Bean Client

In listing 2.1, the `PlaceBid` EJB is called from the web-tier, which would most likely be a JSP or Servlet. For simplicity, we assume that the client is a Servlet named `PlaceBidServlet`. Listing

2.2 shows how the code might look. The Servlet's `service` method calls the `addBid` method to handle the user request to place a bid on an item.

2 Listing 2.2: A simple Servlet client for the PlaceBid EJB

```
package ejb3inaction.example.buslogic;

import javax.ejb.EJB;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class PlaceBidServlet extends HttpServlet {
    @EJB (name="PlaceBid")                               | #1
    private PlaceBid placeBid;
    ...
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws
        ServletException, IOException {
        ...
        placeBid.addBid(bidderId, itemId, bidAmount);    | #2
        ...
    }
}
(annotation) <#1 Inject instance of EJB>
(annotation) <#2 Invoke method on EJB>
```

The `@EJB` annotation in the `PlaceBidServlet` is used to get a reference to the `PlaceBid` Session Bean#1. As you can see, the `name` parameter specifies the name of the EJB to retrieve. When the container sees the `@EJB` annotation, it looks up the EJB reference behind the scenes (remotely if necessary) and injects it into the `placeBid` variable. It is also very important to note the fact that the EJB implementation Class is never used to access EJB references. In our case, we invoke the `addBid()` method through the `PlaceBid` remote interface#2. Believe it or not, this is all there is to writing an EJB client. Other than getting a reference to an EJB, the client code does not look very much different than using a local POJO.

Looking back at Listing 2.1, you will note that the `PlaceBidBean` has no instance variables and the `addBid` method is not dependent on or related to any other Bean method. In fact, the `PlaceBid` EJB represents the extreme case where a Stateless Session Bean contains a single method. Essentially, each invocation of the `PlaceBidBean` is a completely independent action. This is exactly what it means for a Session Bean to be *stateless*. On the flip side, the methods of a Stateful Session Bean are interrelated and often use instance variables.

2.2.2 Using Stateful Beans

Stateless Session Beans model single-step processes, such as placing a bid or viewing an item listing. Stateful Session Beans model multi-step processes, such as ordering an item on ActionBazaar. Because of the inherent complexity involved in the process of ordering an item, it is broken up into multiple small steps. The `PlaceOrderBean` presented in Listing 2.3 implements the item ordering workflow as a Stateful Session Bean. The `setBidder` method is called at the beginning of the workflow by the web application to identify the currently logged-in user as the bidder. Then the `addItem`, `setShippingInfo`, `setBillingInfo` and `confirmOrder` methods are called in sequence from the web-tier. These methods correspond to the distinct ordering steps presented to the winning bidder—adding items to the order, specifying shipping data, adding billing information and

confirming the order after reviewing the complete order. As we alluded to previously, the order confirmation stage results in the following actions: the order record is persisted, the billing step is initiated asynchronously, and the user receives a confirmation when the `confirmOrder()` method returns.

Let's take a look at this chapter's example of a Stateful Session Bean. Listing 2.3 depicts the `PlaceOrderBean` EJB, which tracks the progress of a bidder's order. Also included in Listing 2.3 is the `PlaceOrder` interface used by the `PlaceOrderBean` to place an order in the `ActionBazaar` system.

3 Listing 2.3: OrderBean Stateful Session Bean

```
package ejb3inaction.example.buslogic;

import javax.ejb.Remove;
import javax.ejb.Stateful;
import java.util.ArrayList;
import java.util.List;
import ejb3inaction.example.persistence.Bidder;
import ejb3inaction.example.persistence.BillingInfo;
import ejb3inaction.example.persistence.Item;
import ejb3inaction.example.persistence.Order;
import ejb3inaction.example.persistence.ShippingInfo;

@Stateful(name="PlaceOrder")                                     | #1
public class PlaceOrderBean implements PlaceOrder {             |
    private Bidder bidder;                                       | #2
    private List<Item> items;                                     | #2
    private ShippingInfo shippingInfo;                          | #2
    private BillingInfo billingInfo;                             | #2

    public PlaceOrderBean () {                                    |
        items = new ArrayList<Item>();                          | #3
    }

    public void setBidder(Long bidderId) {                       | #4
        this.bidder = ... Lookup bidder using bidderId ...
    }

    public void addItem(Long itemId) {                           | #4
        Item item = ... Lookup item using itemId ...
        items.add(item);
    }

    public void setShippingInfo(ShippingInfo shippingInfo) {    | #4
        this.shippingInfo = shippingInfo;
    }

    public void setBillingInfo(BillingInfo billingInfo) {        | #4
        this.bilingInfo = billingInfo;
    }

    @Remove                                                       | #5
    public Long confirmOrder() {
        Order order = new Order();
        order.setBidder(bidder);
        order.setItems(items);
        ... Setup the rest of the order data ...
        ... Save order into the database ...
    }
}
```

```

        ... Asynchronously start billing process ...

        return order.getOrderid();
    }
}
...
package ejb3inaction.example.buslogic;

import javax.ejb.Remote;
import ejb3inaction.example.persistence.BillingInfo;
import ejb3inaction.example.persistence.ShippingInfo;

@Remote
public interface PlaceOrder {
    void setBidder(Long bidderId);
    void addItem(Long itemId);
    void setShippingInfo(ShippingInfo shippingInfo);
    void setBillingInfo(BillingInfo billingInfo);
    Long confirmOrder();
}
(annotation) <#1 Mark POJO as Stateful>
(annotation) <#2 State instance variables>
(annotation) <#3 Instantiate state variables>
(annotation) <#4 Business methods>
(annotation) <#5 Remove method>
(annotation) <#6 Remote business interface>
(annotation) <#7 Interface methods>

```

Overall, there is not a huge difference between developing a Stateless and a Stateful Session Bean. As a matter of fact, from the container’s perspective, the only difference that matters is that the `PlaceOrderBean` Class is marked with the `@Stateful` annotation instead of the `@Stateless` annotation#1 and `@Remove` annotation #5. As a developer, you will note that unlike `PlaceBidBean`, `PlaceOrderBean` contains a number of instance variables and each of the business methods uses the instance variables. Although it is not explicit, business methods are expected to be called in a certain order. This is most obvious in the `confirmOrder()` method. This method expects that all the instance variables necessary to complete an order will have been set through previous method calls. In Chapter 4, we will see how Stateful Bean workflow for an order can be explicitly controlled.

It is also important to note the `@Remove` annotation placed on the `confirmOrder` method. This annotation indicates the end of the workflow modeled by `PlaceOrderBean`. To understand why this is important, consider an important side effect of maintaining state. A particular Bean instance must be strictly mapped to a single client for the duration of a workflow. It would be senseless if not dangerous for one client to be able to call the `addItem` method while another client expecting to execute a separate workflow calls the `confirmOrder` method. The `@Remove` annotation tells the container when to end this dedicated “workflow session” mapping so that the Bean can be reclaimed instead of hanging around forever waiting for the next client invocation in the workflow. We will explore this in much greater detail in Chapter 4.

A Stateful Bean Client

Although it is clear that the `PlaceOrder` EJB is called from the `ActionBazaar` web-tier, we will develop the example client as a standalone “test” application, just to add a dash of color. The Stateful Session Bean client in Listing 2.4 goes through a complete test run of the ordering workflow and can be run within an Application Client Container

4 Listing 2.4: Stateful Session Bean Client

```
package ejb3inaction.example.buslogic;

import javax.ejb.EJB;
import ejb3inaction.example.persistence.BillingInfo;
import ejb3inaction.example.persistence.ShippingInfo;

public class PlaceOrderTestClient {
    @EJB(name="PlaceOrder")                                     |#1
    private static PlaceOrder placeOrder;

    public static void main(String [] args) {
        try {
            placeOrder.setBidder(0);
            placeOrder.addItem(0);
            placeOrder.addItem(1);
            ... Setup dummy shipping and billing info ...
            placeOrder.confirmOrder();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
(annotation) <#1 Inject an instance of EJB>
```

As you can see, there is virtually no difference on the client side between in how to access Stateful and Stateless Session Beans. The most significant but critical distinction is that the client can expect the Bean to maintain state on the server-side until calling a method marked with the `@Remove` annotation ends the conversation. To view the complete code sample, and actually run the client, download the `Chapter2.zip` file from <http://ejb3inaction.com>.

This finishes our brief introduction of Session Beans. We are now ready to move on to Message Driven Beans.

2.3 Messaging with Message Driven Beans

Just as Session Beans process business requests from the client, Message Driven Beans process *messages*.. If you have been doing enterprise development, you are probably familiar with at least the basic idea of messaging. In the most basic terms, messaging means communicating between two separate processes, usually across different machines. Java EE messaging follows this basic model on steroids. Most significantly, Java EE makes messaging robust by adding a reliable middleman between the message sender and receiver. This idea is illustrated in Figure 2.2.



2 **Figure 2.2: Java EE adds reliability to messaging by adding a reliable middleman that guarantees the delivery of messages despite network outages, even if the receiver is not present on the other end when the message is sent. In this sense, Java EE messaging has much more in common with the postal service than it does with common RPC protocols like RMI. We will discuss this model in much greater detail in Chapter 5.**

In Java EE terms, the reliable middleman is called a messaging *destination*, powered by an Message Oriented Middleware (MOM). A couple of popular MOM servers are the IBM MQSeries and the SonicMQ. Java EE also standardizes messaging through a well-known JMS (Java Messaging Service) API, which Message Driven Beans rely heavily on. We will discuss messaging, JMS and MDBs (Message Driven Beans) in much greater detail in Chapter 5. For now, this is all you really need.

Messaging has numerous uses in enterprise systems, such as reliable system integration, asynchronous processing, distributed system communication, distributed logging/tracing and so on. In our ActionBazaar example, we enable asynchronous order billing through messaging. To see how this is done, let's revisit the parts of the `PlaceOrderBean` introduced in Listing 2.3 that we omitted, namely the `confirmOrder` method.

2.3.1 Producing a Billing Message

The `PlaceOrderBean` accomplishes asynchronous or “out-of-process” order billing through a nifty technique. Instead of actually going through the billing process in the `confirmOrder` method, the Bean simply generates a message to request that the order billing be started in parallel. As soon as this billing request message is sent to the messaging middleman, the `confirmOrder` method returns the order confirmation to the user, as shown in listing 2.5. The billing request message is sent to a messaging destination named ‘`jms/orderBillingQueue`’. As usual, we have omitted all the code that is not directly related to the current topic.

5 Listing 2.5: PlaceOrderBean Producing JMS Message

```
package ejb3inaction.example.buslogic;

import javax.annotation.Resource;
import javax.ejb.Remove;
import javax.ejb.Stateful;
import javax.jms.*;
import ejb3inaction.example.persistence.Bidder;
import ejb3inaction.example.persistence.BillingInfo;
import ejb3inaction.example.persistence.Item;
import ejb3inaction.example.persistence.Order;
import ejb3inaction.example.persistence.ShippingInfo;

@Stateful(name="PlaceOrder")
public class PlaceOrderBean implements PlaceOrder {
    @Resource(name="jms/QueueConnectionFactory")                | #1
    private ConnectionFactory connectionFactory;

    @Resource(name="jms/orderBillingQueue")                    | #1
}
```

```

private Destination billingQueue;
...
@Remove
public Long confirmOrder() {
    Order order = new Order();
    order.setBidder(bidder);
    order.setItems(items);
    ... Setup the rest of the order data ...
    ... Save order into the database ...
    sendBillingMessage(order);

    return order.getOrderid();
}

private sendBillingMessage(Order order) {                                     | #2
    try {
        Connection connection =                                           | #3
            connectionFactory.createConnection();                          | #3
        Session session =                                                 | #4
            connection.createSession(false,                               | #4
                Session.AUTO_ACKNOWLEDGE);                                | #4
        MessageProducer producer =                                        | #5
            session.createProducer(billingQueue);                          | #5
        ObjectMessage message =                                         | #6
            session.createObjectMessage();                                 | #6
        message.setObject(order);                                         | #7
        producer.send(message);                                           | #8
        producer.close();                                                 | #9
        session.close();                                                  | #9
        connection.close();                                               | #9
    } catch(Exception e) {
        e.printStackTrace();
    }
}
}
}
(annotation) <#1 Inject JMS resources>
(annotation) <#2 Sends JMS message>
(annotation) <#3 Create connection>
(annotation) <#4 Create session>
(annotation) <#5 Create producer for destination>
(annotation) <#6 Create message>
(annotation) <#7 Set message body>
(annotation) <#8 Send message>
(annotation) <#9 Release JMS resources>

```

Not surprisingly, the code to send the message in Listing 2.5 is heavily dependent on the JMS API. In fact, that is all that the code in the `sendBillingMessage` method does. If you are familiar with JDBC, the flavor of the code in the method might seem familiar. The end result of the code is that the newly created `Order` Object is sent as a message to a JMS destination named `'jms/orderBillingQueue'`. We will not deal with the intricacies of JMS now, but will save a detailed discussion of this essential messaging API for Chapter 5. However, we will note the major features of Listing 2.5.

The first thing you should note is that two JMS resources, including the message destination, are injected using the `@Resource` annotation#1. This annotation essentially works the same way as the `@EJB` annotation we saw earlier. It looks up the resources specified through the name parameter and injects it into the EJB. We will discuss more about injecting resources in Chapter 3. Looking at the `sendBillingMessage` method, the relatively mechanical steps of creating a connection and

session are done first (#3 and #4). The `MessageProducer` is then created for the destination to deliver the message#5. The message is then prepared to be sent (#6 and #7). Finally, the message is sent by invoking the `MessageProducer.send` method#8 and all JMS resources are cleaned up. The MDB to bill the order, `OrderBillingMDB`, receives the message from the destination, inspects the `Order` Object embedded in the message and finishes the billing process. This scheme is depicted in Figure 2.3.



3 **Figure 2.3: Asynchronously order billing using MDBs. The Stateful Session Bean processing the order sends a message to the order-billing queue. The billing MDB picks up this message and processes it asynchronously.**

We will now take a look at the `OrderBillingMDB` implementation next.

2.3.2 Using the Order Billing Message Processor

The `OrderBillingMDB` attempts to bill the bidder for the total cost of the order, including the price of the items in the order, shipping, handling, insurance costs and the like. Recall that the `Order` Object in the message contains a `BillingInfo` Object. The `BillingInfo` Object tells the `OrderBillingMDB` how to bill the customer (for example, perhaps by charging a credit card or crediting against an online bank account). After finishing the billing attempt, the MDB notifies both the bidder and seller of the results of the billing attempt. If billing is successful, the seller ships to the address specified in the order. If the billing attempt fails, the bidder must correct and resubmit the billing information attached to the order. Listing 2.6 shows the code for the `OrderBillingMDB`.

6 Listing 2.6: The `OrderBillingMDB`

```

package ejb3inaction.example.buslogic;

import javax.ejb.MessageDriven;
import javax.ejb.ActivationConfigProperty;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.ObjectMessage;
import ejb3inaction.example.persistence.Order;

@MessageDriven(                                     | #1
    activationConfig = {                           | #2
        @ActivationConfigProperty(                 | #2
            propertyName="destinationName",        | #2
            propertyValue="jms/orderBillingQueue") | #2
        }
    )                                               | #1
public class OrderBillingMDB implements MessageListener { | #3
    public void onMessage(Message message) {        | #4
        try {
            ObjectMessage objectMessage = (ObjectMessage) message; | #5
            Order order = (Order) objectMessage.getObject(); | #6
            ...
            bill(order);                             | #7
            ...
        }
    }
}
  
```

```

        notify(order, ...);
        ...
    } catch (Exception e) {
        e.printStackTrace();
    }
}
...
}
(annotation) <#1 Mark as Message Driven Bean>
(annotation) <#2 Specify destination to get messages from>
(annotation) <#3 Implements javax.jms.MessageListener interface>
(annotation) <#4 Implement onMessage, invoked by container>
(annotation) <#5 Cast to correct message type>

```

Like Session Beans, Message Driven Beans are POJOs in EJB3. The `@MessageDriven` annotation marks a Bean class to be an MDB#1. The activation config property nested in the `@MessageDriven` annotation in Listing 2.6 tells the container that the `OrderBillingMDB` processes messages from the JMS destination named 'jms/orderBillingQueue' JMS destination#2. You may have to define additional activation config properties appropriate to your environment. When we send a message to the destination, the container asks the MDB to process it by invoking the *listener method*. Just as Session Bean clients invoke Bean business methods through business interfaces, the container invokes the MDB through a *listener interface*.

As you might guess by looking at Listing 2.6, `javax.jms.MessageListener` is the listener interface for the `OrderBillingMDB`#3. We will explore listener interfaces in much greater detail in Chapter 5. For now, you should note that the `MessageListener` interface defines a single method, `onMessage`, which is implemented in `OrderBillingMDB`#4. The container passes the message retrieved from the JMS destination as the single parameter to the `onMessage` method. The first thing we do with the received message is cast it back to an `ObjectMessage`, since we know this is the message type the `PlaceOrder` Bean generates. We then retrieve the embedded `Order` object, attempt to bill the bidder and send the results of the billing attempt as a notification to both the bidder and the seller. Although we have omitted the code, the notifications are sent as emails using the JavaMail API. We encourage you to download and explore the complete code sample.

Believe it or not, this more or less sums up the process of consuming a message using an MDB. In Chapter 5, we will explore a few more intricacies to Message Driven Beans, as well as some other useful tidbits along the way in Chapters 3 and 6. Now, we move on to the final major EJB topic, the Persistence API.

2.4 Persisting data with EJB 3.0

The EJB 3.0 Java Persistence API (JPA) is used to save application data into relational databases. The API is obviously critical from an application development perspective. In addition, this API is one of the most important innovations introduced in this version of EJB specification. Developing Session and Message Driven Beans has been made significantly simpler through the use of features like metadata annotations. Moreover, interceptors have been introduced to both of these Bean types (see Chapter 3 for more information on interceptors). However, in the end, the underlying Session and Message Driven Bean architecture has not changed much.

The Java Persistence API, on the other hand, is highly inspired by lightweight POJO persistence frameworks like Hibernate and TopLink and represents a truly fresh look at persisting enterprise data. In fact, unlike the Entity Beans model in EJB 2.x, the data saved into the database is not stored

in container-managed EJBs at all. Instead, POJO *Entity Objects* are used to store data. These Entities are mapped to the database table using O-R (Object-Relational) mapping and manipulated using a simple API interface named the *Entity Manager*. As we will soon see, this makes the task of persistence easy and intuitive. A cleanly separated API means that EJB 3.0 Persistence can be used outside a full-scale Java EE container, even in Java SE environments. Actually, the EJB 3.0 Java Persistence API is meant to standardize Java Object Relational Mapping (ORM) solutions like Hibernate, Oracle TopLink and JDO. Under the hood, the EJB3 reference implementation, GlassFish, uses TopLink, while the JBoss AS EJB3 implementation uses Hibernate for its persistence functionality.

Although the Persistence API takes a serious bite out of the complexity in saving enterprise data, persistence is still a non-trivial topic that we will save for the later part of this book—Chapters 7, 8, 9, 10 and 11. Since we save data at almost every step in our example ActionBazaar scenario, all the Beans we have introduced so far use the Persistence API in the omitted portions of code. For sake of simplicity, we will pick a representative example to give you a taste of this new API. To see what EJB 3.0 Persistence looks like, we will revisit the PlaceBid Stateless Session Bean introduced previously.

2.4.1 Working with the Persistence API

You might recall from Listing 2.1 that we omitted the persistence code for actually saving a bid record into the database in the PlaceBid.addBid method. In this section, we will show you how the EJB 3.0 Persistence API can be used to actually save the bid record data. To simplify the code, we will not do any data validation or error handling and will assume ideal application conditions. We will also omit the somewhat complicated aspects of persistence such as saving O-R relationships and embedded objects (Chapters 7, 8 and 9 deal with this). Other persistence intricacies we will avoid for now include transactions (Chapter 6), delete/update/retrieval operations (Chapter 9) and queries (covered in Chapter 10). The code in Listing 2.7 saves the bid record into the BIDS table in the ActionBazaar database when the PlaceBid.addBid() method is invoked by a client.

7 Listing 2.7: Saving a bid record using the EJB 3.0 Persistence API

```
package ejb3inaction.example.buslogic;

import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;
...

@Stateless(name="PlaceBid")
public class PlaceBidBean implements PlaceBid {
    @PersistenceContext(unitName="actionBazaar")           | #1
    private EntityManager entityManager;
    ...
    public Bid addBid(String bidderId, Long itemId,
        Double bidAmount) {
        Bid bid = new Bid();                               | #2

        Bidder bidder = ... Lookup bidder using bidderId ... | #3
        bid.setBidder(bidder);                             | #3
        Item item = ... Lookup item using itemId ...       | #3
        bid.setItem(item);                                 | #3
        bid.setAmount(bidAmount);                         | #3

        entityManager.persist(bid);                       | #4
    }
}
```

```

        return bid;
    }
}
...
package ejb3inaction.example.persistence;

import java.io.Serializable;
import java.sql.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.GenerationType;
import javax.persistence.GeneratedValue;

//Bid Entity Class

@Entity
@Table(name="BIDS")
public class Bid implements Serializable {
    private Long bidId;
    private Double bidAmount;
    private Date bidDate;
    private Item item;
    private Bidder bidder;

    public Bid() {}

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="BID_ID")
    public Long getBidId() {
        return bidId;
    }

    public void setBidId(Long bidId) {
        this.bidId = bidId;
    }

    @Column(name="BID_AMOUNT")
    public Double getBidAmount() {
        return bidAmount;
    }

    public void setBidAmount(Double bidAmount) {
        this.bidAmount = bidAmount;
    }

    @Column(name="BID_DATE")
    public Date getBidDate() {
        return bidDate;
    }

    public void setBidDate(Date bidDate) {
        this.bidDate = bidDate;
    }

    ...
}

```

| #5
| #6

| #7
| #8
| #9

| #9

| #9

```

public Item getItem() {
    return item;
}

public void setItem(Item item) {
    this.item = item;
}

...
public Bidder getBidder() {
    return bidder;
}

public void setBidder(Bidder bidder) {
    this.bidder = bidder;
}
}
(annotation) <#1 Injects instance of EntityManager>
(annotation) <#2 Create new entity instance>
(annotation) <#3 Setup entity data>
(annotation) <#4 Persist entity instance>
(annotation) <#5 Identifying POJO as entity>
(annotation) <#6 Table mapping>
(annotation) <#7 Entity id>
(annotation) <#8 Id value generation>
(annotation) <#9 Column mappings>

```

If you have done any persistence coding using JDBC or EJB 2.x Entity Beans, Listing 2.7 could seem unbelievably simple. All the data being persisted is held in the `Bid` Object, a POJO following the JavaBean naming conventions and containing just a handful of metadata annotations. The real magic of this code lies in the `EntityManager` interface. You can think of this interface as the “interpreter” between the Object Oriented and relational database worlds. The `EntityManager` is smart enough to perform a number of tricks: add a new Object instance into the database as a relational record, read an Object instance saved into the database, update a Object instance stored in the database, and delete an Object instance from the database.

In the code example, we are essentially asking the `EntityManager` to save the newly created `Bid` instance into the database by calling the `persist()` method#4. Note that just like the `@Resource` annotation in Listing 2.5, the `@PersistenceContext` annotation injects an ActionBazaar-specific `EntityManager` into the `PlaceBid` Bean#1. In Chapters 9 and 10, we fully explore all the operations offered through the `EntityManager` interface, including the robust query facility offered by EJB 3.0, the Java Persistence Query Language (JPQL).

The next question to cross your mind should be this: how does the `EntityManager` know what tables and columns to save the `Bid` data into? As you may have guessed by looking at the `Bid` POJO, we have used annotations to tell the `EntityManager` which table the bid record is stored in and what `Bid` Object fields are mapped to which columns in the table. This process of mapping an Object to columns in the database is exactly what O-R is all about. As you will see in Chapter 8, O-R mapping is perhaps the most complex part of the Persistence API. We will not discuss O-R mapping in detail here, but will give you just enough food for thought.

The `@Entity` annotation tells the `EntityManager` that a POJO can be persisted into the database as a unique record#5. You can think of this annotation as the Persistence API counterpart of the `@Stateless`, `@Stateful` and `@MessageDriven` annotations. The `@Table` annotation specifies that the bid record should be saved in the `BIDS` table#6. Similarly, each of the `@Column`

annotations tells the Persistence API which `bid` Object property maps to which column in the BIDS table#9. For example, the `bidId` property maps to the `BIDS.BID_ID` column, the `bidAmount` property maps to the `BIDS.BID_AMOUNT` column and so on. You should note that we have conveniently omitted the mappings for the two custom Objects in the `Bid` Entity, `Item` and `Bidder`. This is because these Objects are Entities themselves and you will need to understand the non-trivial concept of mapping Entity relations to map these properties. To keep this section reasonably digestible, we have chosen to avoid tackling this bit of complexity for now and will defer it to Chapters 7 and 8.

Finally, you should note the `@Id` and `@GeneratedValue` value annotations (#7 and #8). In effect, these annotations specify that `BID_ID` is the primary key of the BIDS table and that the Persistence API should automatically generate a value for this column. We will discuss both of these annotations in much greater detail in Chapter 8. Interpreting all these annotations, the EJB3 persistence provider will generate the following SQL statement in the background when the `Bid` item is saved into the database:

```
INSERT INTO BIDS (BID_ID, BID_DATE, BIDDER, BID_AMOUNT, ITEM_ID)
VALUES (52, NULL, 'eccentric_collector', 20000.50, 100)
```

Most EJB 3.0 Persistence solutions like Hibernate and TopLink allow you to actually view this generated SQL statement in their log files. This can help you understand what is going on behind the scenes and assist in debugging/performance tuning. Believe it or not, this is more or less all there is to Listing 2.7.

This brings us to the end of this brief introduction to the EJB 3.0 Java Persistence API and the end of this whirlwind chapter. At this point, it should be clear to you how simple, effective and robust EJB 3.0 really is, even from a “bird’s eye” view.

2.5 Summary

This chapter introduced the ActionBazaar application, which is used throughout the book. Using a scenario from the ActionBazaar application, we have shown you a cross-section of EJB 3.0 functionality, including Stateless Session Beans, Stateful Session Beans, Message Driven Beans and the EJB 3.0 Java Persistence API. As we stated in the introduction, the goal of this chapter was not to feed you the “guru pill” for EJB 3.0, but rather show you what to expect from this new version the Java enterprise framework. Most of the rest of this book roughly follows the outline of this chapter. Chapter 4 revisits Session Beans, Chapter 5 discusses messaging, JMS and Message Driven Beans, and Chapter 6 explores transactions and security management in EJB. Chapters 7 through 11 are devoted to a detailed exploration of the Persistence API. Chapters 12 through 15 cover advanced topics in EJB.

In the next Chapter, we will shift to a lower gear and talk about some fundamental but critical shared features of EJB 3.0 as preparation to the more targeted, in-depth chapters starting with Chapter 4.