

---

# Sample Application Design and Implementation

**T**HE Java™ Pet Store sample application is a working demonstration of how to use Java™ BluePrints principles in a real application design. This document is an online guide to the design and the implementation of the sample application version 1.3.1. It supplements the book *Designing Enterprise Applications with the Java™ 2, Enterprise Edition, Second Edition*. This document assumes that the reader understands the topics explained in Chapter 11 of the book, *Architecture of the Sample Application*.

The basic architectural features of the sample application change relatively slowly. Each release of the sample application maintains consistent design features such as separating logic and presentation, using servlets for control and JSP™ pages for presentation, using enterprise beans as managed business components, and so on. Yet the sample application source code continues to evolve as new technologies and techniques develop.

While the architecture chapter of the book addresses high-level design features, this document goes deeper into the implementation details of the current sample application release. This document is provided as an online-only resource so that it may be continually extended and updated as both J2EE™ technologies and the sample application evolve.

This document begins with a description of the separate applications that the sample application comprises, describes the modular structure of the pet store application, and provides an in-depth description of several pieces of the modules. An understanding of the topics in Chapter 11 of the book is assumed. An online reference to the book is listed in Resources and References at the end of this document.

## 1.1 High-Level Architectural View

Versions of the sample application before release 1.3 were examples of a monolithic application that handled customer interaction at the Web site, order tracking, and administration.

Real-world enterprise applications are seldom single, monolithic systems. Most enterprise applications must cooperate with multiple data sources and enterprise information systems (EISs). These external systems may be internal information assets, such as legacy databases or enterprise resource planning (ERP) systems. Other external systems may be Web services of business partners. For example, the order fulfillment center is internal to the enterprise, the credit card service is external to the enterprise, and suppliers may be internal (an enterprise warehouse, for example) or external to the enterprise.

The sample application release 1.3 refactors the pet store into separate modules, and also adds new functionality such as the ability to interact with multiple suppliers. The result is a decoupled enterprise architecture that can interoperate with existing data sources and business partners' systems, all built on top of the J2EE platform. The sample application comprises four separate sub-applications that cooperate to fulfill the enterprise's business needs, each of which is a J2EE application:

- **pet store e-commerce Web site (“petstore”)**—a Web application which shoppers use to purchase merchandise through a Web browser
- **pet store administration application (“petstoreadmin”)**—a Web application that enterprise administrators use to view sales statistics and manually accept or reject orders. While petstoreadmin is a Web application, its user interface is a rich client that uses XML messaging, rather than an HTML Web browser
- **order processing center (“OPC”)**—a process-oriented application that manages order fulfillment by providing the following services to other enterprise participants:
  - receives and processes XML documents, via JMS, containing orders from the petstore
  - provides petstoreadmin application with order data using XML messaging over HTTP
  - sends email to customers acknowledging orders using JavaMail™

- sends XML order documents to suppliers via JMS
- maintains purchase order database
- **supplier (“supplier”)**—a process-oriented application that manages shipping products to customers by providing the following services:
  - receives XML order documents from opc via JMS
  - ships products to customers
  - provides manual inventory management through a Web-based interface
  - maintains inventory database

This document concentrates on the architecture of the pet store Web site only. Future documents will offer more in-depth explanation of the other sub-applications.

## 1.2 The Pet Store Web Site

The pet store Web site is the part of the sample application that provides customers with online shopping. Through a Web browser, a customer can browse the catalog, place items to purchase into a virtual shopping cart, create and sign in to a user account, and purchase the shopping cart contents by placing an order with a credit card.

### 1.2.1 Pet Store Web Site Walkthrough

This section presents a look at the Web site from the user’s point of view. The next few subsections show an individual shopper’s “path” through a series of requests to the pet store. If you are already familiar with the pet store Web site, feel free to skip this section.

### 1.2.1.1 Initial Screen

If the pet store is installed on localhost, the user accesses it by using an HTML browser to visit URL `http://localhost:8000/petstore/index.jsp`, shown in Figure 1. This screen allows the user to perform a variety of tasks, such as repopulating the Web site database, running the administration application client, and entering the pet store Web site for shopping. When the user clicks the link labelled *enter the store* the application populates the database with catalog data (unless the data already exist) and redirects the request to URL `/main.screen`. (All URLs listed in this section are relative to the application context root.)

Java<sup>™</sup> Pet Store Demo 1.3

J2EE<sup>™</sup> BluePrints

## Java<sup>™</sup> Pet Store Demo 1.3

The Java<sup>™</sup> Pet Store Demo is a sample application brought to you by the [Java 2 Platform, Enterprise Edition BluePrints \(J2EE<sup>™</sup> BluePrints\)](#) program at [Java Software](#), [Sun Microsystems](#).

This sample application demonstrates how to use the capabilities of the J2EE platform to develop flexible, scalable, cross-platform e-business applications. It comes with full source code and documentation, so you can experiment with J2EE technology and learn how to use it to build your own enterprise solutions.

To start using the demo, [enter the store](#). *If you are using this demo for the very first time, the demo's database will be automatically populated (Be prepared to wait for a while as the database is populated).*

You may also forcefully [repopulate the demo's database](#). (Be prepared to wait for a while as the database is populated.)

You can also use the [administrator client](#).

### About J2EE BluePrints

The J2EE BluePrints program defines the application programming model for the J2EE platform. It provides best practice guidelines and architectural recommendations for real-world application scenarios to enable developers to build portable, scalable, and robust applications using J2EE technology.

The J2EE BluePrints program and the Java Pet Store Demo are showcased in the upcoming second edition of *Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition*. The first edition of this best-selling book, is available at bookstores everywhere, including [DigitalGuru](#) and [Amazon.com](#).

Check out the J2EE BluePrints Web site at <http://java.sun.com/j2ee/blueprints/>.

**Figure 1** Initial Screen (`/index.jsp`)

### 1.2.1.2 Welcome Screen

Once the database is populated, the welcome screen `main.screen` appears, as shown in Figure 2. Like all other pet store screens, this screen HA ARis assembled from a template, and so it has a standard layout. (The layout and template mechanism is discussed in Section 11.4 of the BluePrints book.) The screen allows a user to browse the catalog, sign in, configure account information (if signed in), and switch language and locale. The large image in the center is an image map. Clicking on one of the animals in the map shows the category page for that type of animal. For example, the picture of the dog is linked to URL `/product.screen?product_id=DOGS`.



**Figure 2** Welcome Screen (`/main.screen`)

### 1.2.1.3 Category Screen

The category screen shows all of the products available for a particular category. The product screen for the “DOGS” category is shown in Figure 3. The URL’s GET query string (`category_id=DOGS`) is used by custom tags in the JSP page to produce the appropriate content. Clicking the “Next” link in the bottom right corner of the product list shows the next few items from the list of all products in the category. In this example, clicking the “Chihuahua” link takes the user to the product screen containing all Chihuahuas.



**Figure 3** Category Screen (`/product.screen?category_id=DOGS`)

### 1.2.1.4 Product Screen

The product screen shows all of the items in a particular product. For example, Figure 4 shows all of the chihuahua products in the pet store. Notice that the right column of the list shows the price of the item, and includes a link labelled *Add to Cart*. This link, which also appears on screen for the corresponding item, allows the customer to add the item to the cart without looking at the item details. It's often a good idea to provide access to the same function in more than one place, providing more sophisticated users with a "shortcut". This technique makes the user experience faster and cuts down on page requests. In this example, the customer clicks the link *Adult Male Chihuahua*, which displays the item page for that item.



The screenshot displays the Java™ Pet Store web application. At the top left is a green parrot icon. The main header reads "Java™ Pet Store" and "J2EE™ BluePrints Sample Application". To the right of the header is a search bar, a "Search" button, and links for "Account", "Cart", and "Sign out". Below the header are three main sections: "Pets", "Items for this Product", and "My List".

**Pets**

- [Birds](#)
- [Cats](#)
- [Dogs](#)
- [Fish](#)
- [Reptiles](#)

**Items for this Product**

<a href="#">Adult Male Chihuahua</a>	\$125.50
Little yapper	<a href="#">Add to Cart</a>
<a href="#">Adult Female Chihuahua</a>	\$155.29
Great companion dog	<a href="#">Add to Cart</a>

**My List**

- [Iguana](#)
- [Rattlesnake](#)

Below the main content area, there is a footer section with the following text:

The Java Pet Store Demo is a fictional sample application from the J2EE BluePrints. For more information, visit the J2EE BluePrints Web site at <http://java.sun.com/j2ee/blueprints/>.

[J2EE BluePrints](#) | [Java Software](#) | [Sun Microsystems](#)

© 2001 Sun Microsystems Inc. All rights reserved. Use is subject to license terms.

**Figure 4** Product Screen (/items.screen?product\_id=K9-CW-01)

### 1.2.1.5 Item Screen

The item screen shows detailed information about an individual item for sale, including a photo if one is available. The item screen for an Adult Male Chihuahua appears in Figure 5. The *Add to Cart* link, when clicked, adds an order for the item to the shopping cart, and then shows the shopping cart contents.

The screenshot shows the Java™ Pet Store interface. At the top left is a green parrot icon. The main header reads "Java™ Pet Store" and "J2EE™ BluePrints Sample Application". On the top right, there is a search bar, a "Search" button, and links for "Account", "Cart", and "Sign out". Below the header, there are two navigation boxes: "Pets" on the left with links for "Birds", "Cats", "Dogs", "Fish", and "Reptiles"; and "My List" on the right with links for "Iguana" and "Rattlesnake". The main content area features a photo of a brown Chihuahua puppy. To the right of the photo, the text reads "Adult Male Chihuahua", "List Price: \$125.50", and "Little yapper". Below the photo is a blue "Add to Cart" link. At the bottom of the page, there is a disclaimer: "The Java Pet Store Demo is a fictional sample application from the J2EE BluePrints. For more information, visit the J2EE BluePrints Web site at <http://java.sun.com/j2ee/blueprints/>." Below the disclaimer are links for "J2EE BluePrints", "Java Software", and "Sun Microsystems", followed by the copyright notice: "© 2001 Sun Microsystems Inc. All rights reserved. Use is subject to license terms."

**Figure 5** Item Screen (/item.screen?item\_id=EST-26)

### 1.2.1.6 Cart Screen

The cart screen, shown in Figure 6, lists the items currently in the cart, allows the customer to change the quantity of each item ordered, and shows a title. It also includes a link to remove the item from the cart, and a link *Proceed to Checkout* which, when clicked, shows the order information screen if the user is signed on. If the user is not signed on, the signon screen is shown instead.

**Java™ Pet Store**  
J2EE™ BluePrints Sample Application

Search  
Account | Cart | Sign out  
🇺🇸 🇯🇵

---

**Pets**  
[Birds](#)  
[Cats](#)  
[Dogs](#)  
[Fish](#)  
[Reptiles](#)

**Your Shopping Cart**

<a href="#">Adult Male Chihuahua</a>	<a href="#">Remove</a>	<input type="text" value="1"/>	@ \$125.50
<input type="button" value="Update Cart"/>		<b>Subtotal:</b> \$125.50	
<a href="#">Proceed to Checkout</a>			

**My List**  
[Iguana](#)  
[Rattlesnake](#)

---

The Java Pet Store Demo is a fictional sample application from the J2EE BluePrints. For more information, visit the J2EE BluePrints Web site at <http://java.sun.com/j2ee/blueprints/>.

[J2EE BluePrints](#) | [Java Software](#) | [Sun Microsystems](#)  
© 2001 Sun Microsystems Inc. All rights reserved. Use is subject to license terms.

**Figure 6** Cart Screen (/cart.do?action=purchase&itemId=EST-26)

### 1.2.1.7 Signon Screen

The signon screen (Figure 7) allows an existing customer to sign in as an existing user, and a new customer to create an account. An existing customer enters a user name and password, and the application displays the Order Information screen shown in Figure 9. A new customer enters a user name and password and clicks the button *Create New Account*. The application creates a user with the requested password. If user creation succeeds, the application displays the Account Information screen.

**Java™ Pet Store**  
J2EE™ BluePrints Sample Application

[Account](#) | [Cart](#) |   | [Sign In](#)

---

**Pets**

- [Birds](#)
- [Cats](#)
- [Dogs](#)
- [Fish](#)
- [Reptiles](#)

**Sign In**

Are you a returning customer?

<p><b>Yes.</b></p> <p>User Name: <input type="text" value="j2ee"/></p> <p>Password: <input type="password" value="****"/></p> <p><input type="button" value="Sign In"/></p>	<p><b>No. I would like to sign up for an account.</b></p> <p>User Name: <input type="text"/></p> <p>Password: <input type="password"/></p> <p>Password (Repeat): <input type="password"/></p> <p><input type="button" value="Create New Account"/></p>
---	--

---

The Java Pet Store Demo is a fictional sample application from the J2EE BluePrints. For more information, visit the J2EE BluePrints Web site at <http://java.sun.com/j2ee/blueprints/>.

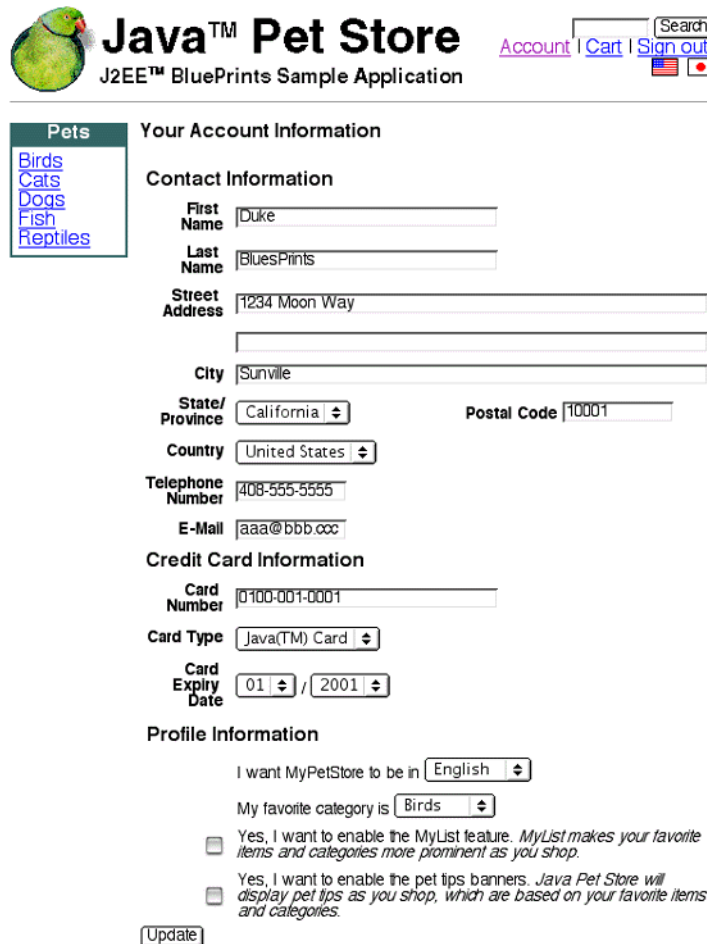
[J2EE BluePrints](#) | [Java Software](#) | [Sun Microsystems](#)

© 2001 Sun Microsystems Inc. All rights reserved. Use is subject to license terms.

**Figure 7** Signon Screen (/signon.screen)

### 1.2.1.8 Account Information Screen

The account information screen, shown in Figure 8, collects information about a new customer, including contact information, a credit card, and personal preferences. This is also the screen displayed whenever the customer clicks the *Account* link at the top right corner of the screen (beneath the *Search* box). Clicking the *Update* button directs the browser to a page that summarizes the information entered.



**Java™ Pet Store**  
J2EE™ BluePrints Sample Application

[Account](#) | [Cart](#) | [Sign out](#)

[US](#) [JP](#)

**Pets**

- [Birds](#)
- [Cats](#)
- [Dogs](#)
- [Fish](#)
- [Reptiles](#)

**Your Account Information**

**Contact Information**

First Name

Last Name

Street Address

City

State/Province  Postal Code

Country

Telephone Number

E-Mail

**Credit Card Information**

Card Number

Card Type

Card Expiry Date  /

**Profile Information**

I want MyPetStore to be in

My favorite category is

Yes, I want to enable the MyList feature. *MyList makes your favorite items and categories more prominent as you shop.*

Yes, I want to enable the pet tips banners. *Java Pet Store will display pet tips as you shop, which are based on your favorite items and categories.*

**Figure 8** Account Information Screen (/createuser.do)

### 1.2.1.9 Order Information Screen

The Order Information screen allows the user to enter billing and shipping address. Default values for the addresses come from the contact information for the currently signed-in customer. This information is transmitted to the application when the user clicks *Submit*. The application creates a new order, sends it to the order processing center, and displays the Order Complete screen.

**Java™ Pet Store** Account | Cart | Sign out Search  
J2EE™ BluePrints Sample Application

**Pets**  
Birds  
Cats  
Dogs  
Fish  
Reptiles

**Your Account Information**

**Billing Information**

First Name: XYZ  
Last Name: ABC  
Street Address: 1234 Anywhere Street  
Unit: 555  
City: Palo Alto  
State/Province: CA Postal Code: 10001  
Country: United States  
Telephone Number: 555-555-5555  
E-Mail: aaa@bbb.ooo

**Shipping Information**

First Name: XYZ  
Last Name: ABC  
Street Address: 1234 Anywhere Street  
Unit: 555  
City: Palo Alto  
State/Province: CA Postal Code: 10001  
Country: United States  
Telephone Number: 555-555-5555  
Submit

The Java Pet Store Demo is a fictional sample application from the J2EE BluePrints. For more information, visit the J2EE BluePrints Web site at <http://java.sun.com/j2ee/blueprints/>.

[J2EE BluePrints](#) | [Java Software](#) | [Sun Microsystems](#)  
© 2001 Sun Microsystems Inc. All rights reserved. Use is subject to license terms.

**Figure 9** Order Information Screen

### 1.2.1.10 Order Complete Screen

The Order Complete screen verifies to the user that the order has been placed. The screen includes the order number.

 **Java™ Pet Store**  
J2EE™ BluePrints Sample Application

Search  
Account Cart  
Sign out  
 

---

**Pets**  
[Birds](#)  
[Cats](#)  
[Dogs](#)  
[Fish](#)  
[Reptiles](#)

**Your Order is Complete**

---

Your order Id is 10012

You should receive a confirmation e-mail soon at  
aaa@bbb.ccc

Thank you for shopping with the Java Pet Store Demo.

---

**My List**  
[Iguana](#)  
[Rattlesnake](#)

---

The Java Pet Store Demo is a fictional sample application from the J2EE BluePrints. For more information, visit the J2EE BluePrints Web site at <http://java.sun.com/j2ee/blueprints/>.

---

[J2EE BluePrints](#) | [Java Software](#) | [Sun Microsystems](#)  
© 2001 Sun Microsystems Inc. All rights reserved. Use is subject to license terms.

**Figure 10** Order Complete Screen

## 1.2.2 Pet Store High-Level Design Choices

The pet store Web site reflects several design choices that have a global impact on application behavior and performance. Chapter 11 of the BluePrints book discusses high-level design choices in some detail. Among these choices are:

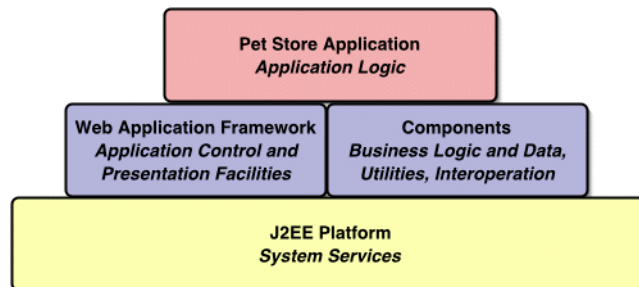
- **Use of an application framework.** Smaller applications often either use no application framework, or manage requests using simple dispatching mechanisms and loose coding conventions. Larger applications often benefit from an MVC application framework like the WAF that ships with the sample application. The consistent structure and MVC functional separation imposed by the framework makes applications more reliable and easier to maintain and extend. Application components developed for a framework are often more reusable, as well. For these reasons, the pet store Web site is designed around an MVC application framework.
- **Web-tier business logic vs. enterprise beans components.** Many applications implement all of their business logic in Web-tier classes. More complex and larger-scale applications often choose enterprise beans technology to provide scalability, reliability, a component-based development model, and common horizontal services such as persistence, asynchronous communication, and declarative transaction and security control. The pet store Web site uses enterprise beans because of these benefits.
- **Local vs. distributed architecture.** Enterprise beans clients in a distributed architecture access enterprise beans components through remote interfaces. While remote enterprise beans improve scalability and availability, the high cost of remote communication makes them appropriate mostly for coarse-grained operations. Local enterprise beans reside in the same JVM™ as their clients, so they avoid costly remote procedure calls. Local enterprise beans provide high-performance access to fine-grained business logic, while maintaining the high-level services of an enterprise beans container. The pet store Web site uses the local client view of enterprise beans to improve performance and simplify development.
- **Declarative vs. programmatic transaction control.** J2EE transactions can be controlled programmatically in code or declaratively in deployment descriptors. Declarative transactions are easier to create and manage, while programmatic transactions offer a higher degree of control. The pet store Web site uses programmatic transaction control to ensure data consistency when generating

views, and declarative transaction control when updating data in the EJB™ tier.

- **Synchronous vs. asynchronous communication.** Synchronous communication is most useful when an operation can produce a result in a reasonable amount of time. Asynchronous communication is more complex to implement and manage, but is useful for integrating high-latency, unreliable, or parallel operations. Most applications use a combination of synchronous and asynchronous communication. For example, the pet store Web site accesses its catalog synchronously, because accessing a catalog is a fast operation. The Web site transmits purchase orders asynchronously because orders may take a long while to complete, and the order processing center may not always be available.

### 1.2.3 Pet Store Web Site Structure

The pet store Web site is built on top of other services, as shown in Figure 11. Pet store Web site code is built on top of the Web Application Framework (“WAF”), an application framework that is shipped with the sample application. J2EE applications can use WAF facilities to control application screen flow, generate views, and activate business components that perform the application’s business functions. Both the business components and the WAF themselves depend on J2EE platform technologies.



**Figure 11** The Pet Store Is Built On Top Of Other Services

The WAF provides a number of services that most Web applications require, including request filtering and dispatching, templated view generation, a set of reusable custom tags, and screen flow control. Code that implements functions

specific to the pet store Web site, shown as the top box in Figure 11, uses hooks into the WAF to extend the WAF's default behavior.

Like the application logic, most the components shown in Figure 11 are application-specific: they represent business data and operations on those data. Entity beans represent business entities such as Customer, Address, and Account. Session beans handle such application functions as signing a user on and off of the system and managing the shopping cart. Other session beans provide utility functions such as generating unique identifiers. Traditional JavaBeans™ components serve as value object classes to communicate between components within the application, and XML document classes are used to interoperate with the order processing coordinator.

The application-specific code shown in Figure 11 consists mostly of “glue” code that ties user interface events to back-end business component operations. For example, when a user requests that the pet store application add an item to the shopping cart, the WAF activates application code that adds the item to the shopping cart component.

### 1.2.4 Pet Store Modular Design

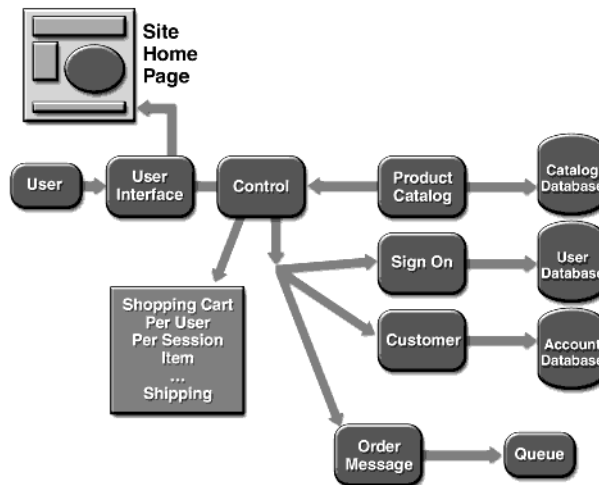
As described in Chapter 11 of the BluePrints book, the pet store was developed as a collection of separate functional modules with well-defined interfaces. Modular development decouples components from one another. This decoupling eases maintenance, simplifies parallel development, and provides opportunities for incorporating third-party components. The pet store comprises the following modules:

- **Control module**—the control module dispatches requests to business logic, controls screen flow, coordinates component interactions, and activates user signon and registration. The control module is implemented by the WAF and application-specific WAF extensions.
- **Shopping cart module**—the shopping cart tracks the items a user has selected for purchase
- **Signon module**—the signon module requires a user to sign on before accessing certain screens, and manages the sign on process
- **Messaging module**—the messaging module asynchronously transmits purchase orders from the pet store to the OPC
- **Catalog module**—the catalog module provides a page-based view of the cat-

alog based on user search criteria

- **Customer module**—the customer module represents customer information: addresses, credit cards, contact information, and so on

The figure from Chapter 11 of the BluePrints book that shows how these modules are related appears in Figure 12.



**Figure 12** Pet Store Functional Modules

As the figure shows, the user accesses the pet store using the browser as a user interface. The control module mediates all communication and execution in the pet store. Each user session has a shopping cart object that it uses to track any purchases.

### 1.2.5 Module Design Discussions

Each pet store module has different requirements from the others. This section describes the requirements, design, and implementation of each module.

### 1.2.5.1 Control Module

The pet store control module is composed of the WAF plus application-specific classes and files that extend the WAF. It coordinates the actions of all of the other modules, and is the only module that interacts directly with the user.

#### Control Module Requirements

The control module forms the framework underlying the application, so it naturally has the most high-level requirements. Extensibility and maintainability are prime considerations in this module. The control module must be extensible because all real-world enterprise applications change constantly. Because the control module plays a role in virtually every interaction, its code must be well-structured to avoid complexity-related maintenance problems. The requirements of the control module are:

- **The module must handle all HTTP requests for the application.** This module controls a Web application, and interacts with a user who sends HTTP service requests. The control module is responsible for classifying and dispatching each of these requests to the other modules.
- **HTTP responses may be of any content type.** Web application developers do not want to be limited to just textual content types. The control module must also be able to produce binary responses.
- **Business logic must be easily extensible.** Enterprise applications are always changing and growing as business rules and conditions change. A developer familiar with the control module should be able easily to add new functionality with minimal impact on existing functions.
- **New views must be easy to add.** Most business logic changes imply new user views, so developers must be able to create new views easily.
- **Module must provide application-wide control of look and feel.** Manually editing hundreds or dozens of user views is not a practical way to manage application look and feel. The control module must provide a way for application screen layout and style to be controlled globally.
- **Application must be maintainable even as it grows.** The control module must be structured so that new functions added over time do not result in a thicket of unmaintainable “spaghetti” code. It should be possible to easily understand the control of even an application with several hundred business

classes.

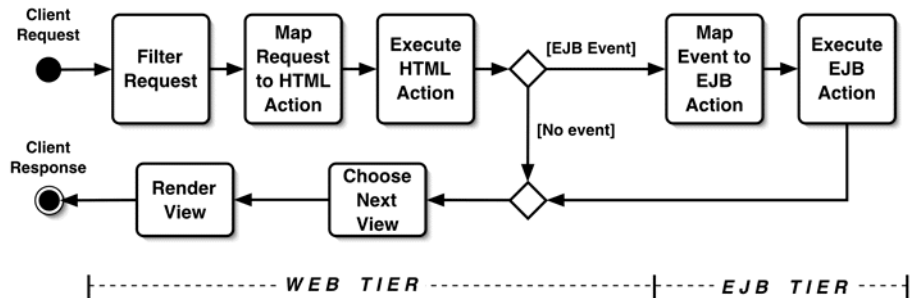
- **Module must allow fine-grained technology selection.** Each model or view component must have access to all J2EE services.
- **Application-wide functionality must be easy to add.** New application requirements occasionally apply to every operation or view in an application. The control module must be structured so that such requirements can be easily met.
- **Must not compromise existing J2EE platform benefits.** The J2EE platform offers such benefits as scalability, portability, declarative transaction and security control, and freedom of technology choice, among others. The control module must maintain these platform goals.

### Control Module Design

Section 4.4 of the Web tier chapter of the BluePrints book discusses Web application framework design at a high level. The sample application's Web Application Framework (WAF) are a specific implementation of the concepts in that chapter.

One of the most important application design decisions is whether or not to use an application framework. The Java BluePrints recommendation is to use an existing, established application framework rather than creating one from scratch. See "Resources and References," at the end of this document, for more on the benefits and drawbacks of using an application framework.

The Web tier service cycle involves four steps: interpreting a request, executing business logic, selecting the next view, and generating the selected view. The activity diagram in Figure 13 presents the service cycle in a bit more detail, showing the actions that the WAF performs in response to each HTTP request (POST or GET).



**Figure 13** Web Application Framework Service Cycle

The WAF receives requests from clients, services the request, and selects and renders the next view to produce the response. Each time the WAF receives a request, it performs the following operations:

- **Filter request**—optional servlet filters intercept requests to provide uniform services (such as security, logging, or encoding enforcement) for every request
- **Map request to HTML action**—for each request, the WAF identifies an *HTML action* class that implements the requested service. The HTML action class is an application-specific class written by a component or application developer.
- **Execute HTML action**—the WAF executes the selected HTML action, which performs the Web tier portion of the requested business operation
- **Choose next view**—the WAF selects the next view to display based on the current view, the results of the HTML action, and possibly other state.
- **Render view**—the WAF generates the selected view and delivers it to the client

An HTML action may return a serializable *EJB event*, which encapsulates a request and arguments to be executed in the EJB tier. When an HTML action returns an EJB event, the WAF performs the following actions in the EJB tier:

- **Map event to EJB action**—the EJB event contains the name of an *EJB action*, which is a developer-defined class containing EJB-tier business logic. The EJB event itself indicates the type of EJB action to create and execute.
- **Execute EJB action**—the WAF executes the EJB action, passing the EJB event as an argument. The action retrieves arguments from the event and manipulates enterprise beans or other data resources to perform the desired business operation.

The WAF performs this same sequence of operations for every request (except requests intercepted by a servlet filter). Each of these steps may be extended by an application developer to provide application-specific functionality.

The control module is composed of several components that work together to meet the control module requirements listed above. These components are:

- A *Front Controller servlet* receives and processes every HTTP request (except

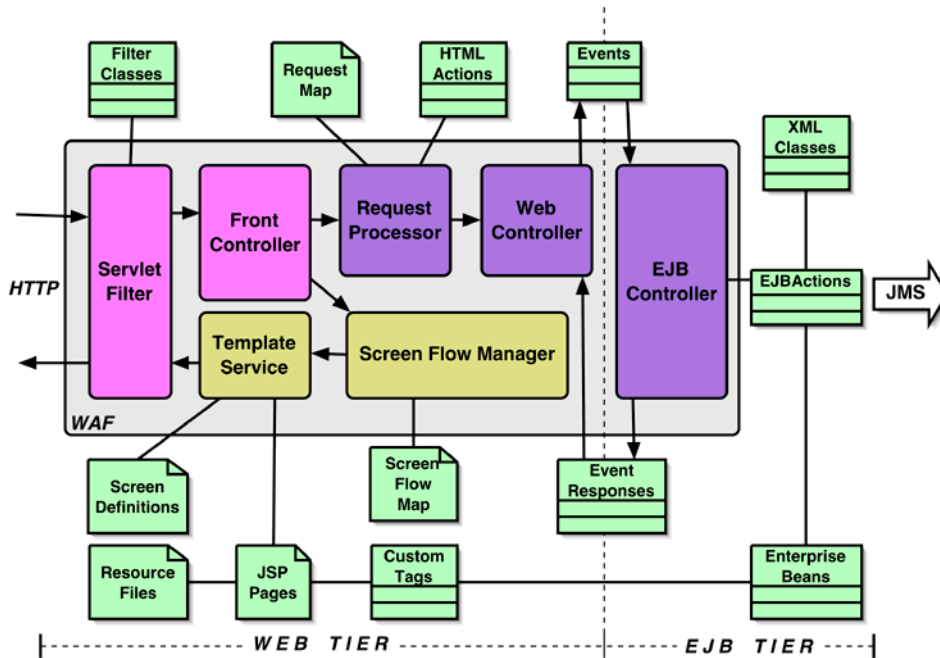
those intercepted by servlet filters). The Front Controller servlet coordinates all other control components to dispatch requests and to select and generate views. Because it is a servlet, it can create content of any type, including binary content types. The Front Controller uses the following components to do its job:

- A *Request Processor* class maps request URLs to actions in the Web tier. Web-tier action classes make it easy for developers to add Web-tier functionality incrementally, while maintaining an easy-to-understand application design.
- A *Web Controller* dispatches events to the EJB controller (see below).
- A *Screen Flow Manager* determines what screen to display after each request is serviced. Screen flow control is defined declaratively in an XML configuration file, so new views and screen flows are easy to add.
- An *EJB controller* in the EJB tier interprets and executes events as EJB actions in the EJB tier. Events and EJB actions make it easy to add new business logic while maintaining a clear design. The choice of HTML actions and EJB actions allow a developer to choose the most appropriate tier for new functionality. Action classes place no restrictions on J2EE technologies available in each tier.
- *Servlet filters* may be placed in front of the Front Controller servlet to add functionality that applies to all requests.
- A *template service* servlet assembles multiple content sources, often JSP pages, into a template that provides a common look and feel to all views. Using JSP pages to create most content makes new views easy to construct.

### Control Module Implementation

This section explains the various ways a developer may extend the WAF to produce an application, and points out design patterns used in the WAF. For more on each pattern, see the corresponding section of the J2EE patterns book, listed in “Resources and References,” at the end of this document.

Figure 14 is a high-level block diagram that shows how the WAF interacts with application-specific components and code.



**Figure 14** An Application As An Extension of the WAF

The application developer can add application functions to each of the steps in the Web request service cycle (described in the section “Control Module Design,” above) as follows:

- Filter request**—each HTTP request optionally may be intercepted by a chain of one or more developer-defined servlet filters. Application developers may write servlet filters (classes that implement `javax.servlet.Filter`) to “wrap” existing servlets and JSP pages with new functionality, or to apply uniform operations (such as logging or security) to all requests or responses. The pet store uses two such servlet filters: `EncodingFilter`, which ensures that request and response encoding match; and `SignOnFilter`, which enforces security and performs user signon.

For more on using servlet filters, and on the Intercepting Filter pattern, see

“Resources and References,” at the end of this document.

- **Centralize control**—the Front Controller servlet (`MainServlet`) processes all requests coming into the pet store Web site. It handles request dispatch, screen flow, and view generation, though it delegates most of these tasks to other classes. An application developer may extend a Front Controller either by wrapping it in a servlet filter (discussed previously), by replacing the components the Front Controller uses (such as the request dispatcher or screen flow manager), or by subclassing and overriding some of the Front Controller’s methods.

For more on using a Front Controller in a J2EE design, see “Resources and References,” at the end of this document.

- **Map request to HTML action and execute HTML action**—the WAF’s `RequestProcessor` class activates a specific business operation based on the URL of each request. An application developer or deployer defines an XML file called a *request map* that maps each request URL to a specific *HTML action*. An HTML action is a developer-defined class (implementing interface `com.sun.j2ee.blueprints.waf.controller.web.action.HTMLAction`) that performs Web-tier business operations. Each time the `RequestProcessor` receives a request, it uses the URL to look up the corresponding HTML action class in the request map, creates an instance of that class, and executes the action by invoking the action’s `perform` method.

A developer defines an HTML action by implementing interface `HTMLAction`, which contains the following methods:

- `public void doStart(HttpServletRequest req)`—initializes the action
- `public Event void perform(HttpServletRequest req)` throws `HTMLActionException`—performs the action’s business function. If it throws an exception, the WAF directs the request to an appropriate error page. The action returns either `null` or an `Event` object that represents a command to be executed in the EJB tier.
- `public void doEnd(HttpServletRequest req, EventResponse evr)`—any logic to be performed after the action is performed; called only if `perform` does not throw an exception

Abstract utility class `HTMLActionSupport` provides empty implementations of all of `HTMLAction`’s methods (other than `perform`, which subclasses must define). Developers may subclass this utility class and override only the desired

methods.

For example, when the client browser POSTs a request to the pet store relative URL “/order.do”, the request processor finds the HTML action corresponding to that URL in the request map (`OrderHTMLAction`) and executes it. The HTML action then performs the business operation of ordering the items in the shopping cart.

- **Map event to EJB action and execute EJB action**—an HTML action may optionally return an `Event` to the request processor to request operations in the EJB tier. To access the EJB tier, a developer defines two classes: an `Event` (implementing interface `com.sun.j2ee.blueprints.waf.event.Event`) and an `EJBAction` (implementing interface `com.sun.j2ee.blueprints.waf.controller.ejb.action.EJBAction`). An `Event` encapsulates a request and its arguments for execution in the EJB tier, while an `EJBAction` implements the EJB-tier business logic for the request.

The WAF’s `RequestProcessor` (in the Web tier) sends any event returned by an HTML action directly to the WAF’s Web controller, which passes that event to the EJB controller. The EJB controller, which is a stateful session bean, uses the data encapsulated in the `Event` to create and execute the `EJBAction`. Application developers may therefore implement business logic using an `HTMLAction` in the Web tier, or using an `EJBAction` in the EJB tier, or both working together by way of an event. An `EJBAction` may optionally use an `EventResponse` (subclassing `com.sun.j2ee.blueprints.waf.event.EventResponse`) to return response data to the Web tier. Because both an `Event` and an `EventResponse` may be passed through a remote interface (if the EJB controller is remote), both must be `Serializable`.

A developer usually defines an `EJBAction` class by subclassing abstract utility class `EJBActionSupport`, defining `perform` and possibly overriding the other methods shown here:

- `public void doStart()`—does any setup necessary for performing the business operation
- `public EventResponse perform(Event ev)` throws `EventException`—performs the business operation, and returns either `null` or an `EventResponse` that represents the result of the operation. Any `EventException` thrown is propagated back to the Web tier, usually resulting in the display of an error page.
- `public void doEnd()`—performs any post-processing before result is returned to client

In the ordering example discussed previously, `OrderHTMLAction` creates an order by returning an `OrderEvent` containing order data (shipping and billing information, credit card number, etc.) to the `RequestProcessor`. The `RequestProcessor` forwards the event to the EJB controller. The EJB controller uses the event to create and execute an `OrderEJBAction`, which actually creates the order by sending an XML message via JMS to the Order Processing Center for fulfillment. It returns an `OrderEventResponse`, which contains an order confirmation. The WAF stores the `OrderEventResponse` in session scope for use by the “order completed” JSP page.

- **View selection**—after executing business logic, the WAF selects the next view (or “screen”) to display. The application developer or deployer specifies the next screen to display in a *screen flow map*, which is an XML file that configures the WAF’s screen flow manager. The screen flow map indicates the name of the next screen to display for each request URL. Usually, the relationship between a request and the next screen to display is static, so it is indicated directly with an XML attribute. When necessary, a developer can programmatically determine the next screen to display using a WAF hook called a screen flow handler (interface `com.sun.j2ee.blueprints.waf.controller.web.flow.FlowHandler`). The developer writes the flow handler and then associates it with the request URL in the screen flow map.

For example, the pet store screen flow map defines `order_complete.screen` as the screen to display after receiving the request URL `/order.do`, as shown in Code Example 1. Note that the WAF specifies the screen flow map and the request map in a single file, `mappings.xml`. A `url-mapping` element in the `mappings.xml` file defines both the next screen to display and the action to perform in response to a particular request URL.

```
<url-mapping url="order.do" screen="order_complete.screen">
  <action-class>
    com.sun.j2ee.blueprints.petstore.controller.web.actions.OrderHTMLAction
  </action-class>
</url-mapping>
```

**Code Example 1** Excerpt From the Pet Store File `mappings.xml`

- **Screen assembly and response generation**—the WAF includes a set of reusable custom tags for building JSP pages, and a templating service to provide

uniform screen layout. An application developer creates an XML *screen definitions file* that defines all of the screens in the application. The screen definitions file indicates the path to the template file (relative to the Web application context root; for example, `/template.jsp`), and then defines a collection of screens. Each screen consists of a name and a collection of elements that bind named areas of the template to specific text strings or JSP pages. (For more on the templating service, see “Resources and References,” at the end of this document.)

The templating service is a servlet that is reusable as a completely separate component in any Web application. The pet store screen flow manager determines the name of the next screen to display, and then forwards the HTTP request to a URL composed of the templating service’s context root, the screen’s name, and the suffix `.screen`. A servlet mapping routes the request to the templating service.

JSP pages may be composed using templates or used individually. The WAF provides a set of reusable custom tags that the developer may use to create cleanly-structured JSP pages.

### **Transaction Control**

The control module must read and modify data resources transactionally to ensure data consistency. Transactional data resources include CMP entity beans, JDBC™ database connections, connectors that support transactions, and JMS providers. The pet store Web site control module ensures transactional access to data resources.

There are two places in the Web tier service cycle (shown in Figure 13) that data resources may be accessed: in an EJB action (a class that implements `EJBAction`), and in the Web-tier template service. The WAF handles transactions declaratively in the EJB tier, and programmatically in the Web tier.

In the EJB tier, EJB actions activated by the EJB controller often use enterprise beans, which in turn may access data resources or participate in JMS messaging. To ensure data consistency, any manipulation of data resources by EJB actions must always be transactional.

The WAF EJB controller uses declarative transaction control to ensure transactional access to data resources. The EJB-tier deployment descriptor `ejb-jar.xml` defines the transaction attribute `Required` for any call to method `ShoppingControllerEJB.processEvent`, which is the method that clients use to execute EJB-tier events. Code Example 2 shows an excerpt from the deployment descriptor that defines the transaction attribute. Whenever `processEvent` is called,

the EJB container starts a new transaction (unless one already exists), and any manipulation of data resources occurs under the scope of that transaction.

```

<container-transaction>
  <method>
    <ejb-name>ShoppingControllerEJB</ejb-name>
    <method-intf>Local</method-intf>
    <method-name>processEvent</method-name>
    <method-params>
      <method-
param>com.sun.j2ee.blueprints.waf.event.Event</method-param>
    </method-params>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>

```

## Code Example 2 The EJB Controller Uses Declarative Transaction Control

In the Web tier, the template service assembles multiple JSP pages and Web resources into a single composite view. Much of the data in application views is generated by custom tags, which in turn use data resources such as entity beans with CMP, JDBC connections, connectors, or JMS providers. To ensure isolation, all access to such data resources for a single composite view must occur under the scope of a transaction.

There is no way to automatically start a transaction in response to executing a Web-tier method, as can be done with an enterprise bean method. Therefore, the pet store uses programmatic transaction control in the template service servlet `TemplateServlet`. `TemplateServlet` begins a `UserTransaction` before it forwards a request to the template JSP page, and ends the transaction after the forward has completed, as shown in Code Example 3.

```

boolean tx_started = false;
UserTransaction ut = null;

// Look up the UserTransaction object
InitialContext ic = new InitialContext();
ut = (UserTransaction) ic.lookup("java:comp/UserTransaction");
ut.begin(); // start the transaction.

```

```

    tx_started = true;
    ...
    context.getRequestDispatcher(templateName).forward(request,
response);
    ...
    if (tx_started && ut != null) {
        ut.commit();
    }
}

```

### **Code Example 3**     `TemplateServlet` Uses A Programmatic Transaction Control

The template servlet provides an excellent example for when programmatic transaction control is preferable over declarative transaction control. If transaction control were declarative, every method call on an enterprise bean or other data resource, including such fine-grained operations as reading and setting attributes, would create, begin, and commit within a separate transaction. Not only would such an arrangement be slow, but views might also reflect inconsistent data, because the data might change between transactions.

The `TemplateServlet` “wraps” multiple fine-grained enterprise bean operations in a single transaction, so only a single transaction is created, improving performance. In addition, reads within the single transaction are repeatable, maintaining isolation from other transactions, and data consistency within the view. All of the data in the view correspond to a “snapshot” of the data as it existed when the `UserTransaction` began.

One possible alternative approach might have been to do all fine-grained enterprise bean access behind a session facade which all clients would be required to use. The session facade could then use declarative transaction control on its methods. But a session facade implies coarse-grained enterprise bean interfaces, which would eliminate the fine-grained benefit of local interfaces. So for the Web tier template service, programmatic transaction control is the superior design choice.

It is possible to access transactional data resources in HTML actions (subclasses of `HTMLAction`), but applications that do so must manage their own transactions. If your design uses enterprise beans, it’s usually better to confine transactional data access to a component in the EJB tier and access it from the Web tier through a component interface. If you are not using enterprise beans, or

if you still need to do transactional work in the Web tier, use programmatic transaction control with the `UserTransaction` interface.

### 1.2.5.2 User Signon and Customer Registration Module

The user signon and customer registration process is controlled by the `SignOnFilter` component. How this component works is interesting because it provides a specific example of how developer-defined classes and configuration information plug application-specific functionality into the WAF. The filter itself is an example of a servlet filter. It provides an example of how to use the screen flow and request map, and how to implement standard interfaces to implement business logic in the Web and EJB tiers.

#### User Signon and Customer Registration Module Requirements

Based on the pet store enterprise's business rules and customer needs, the requirements for user signon and customer registration in the pet store are:

The pet store supports two kinds of customers: registered customers and anonymous customers:

- An anonymous customer may:
  - browse the pet store catalog
  - place catalog items in a virtual shopping cart
  - change locale
  - become a registered customer by providing a user name, password, and several pieces of personal information
- A registered customer may also:
  - sign on to the pet store using the customer's user name and password
  - sign off from the pet store; subsequent requests require explicit signon
- Only if signed on, a registered customer may also:
  - update personal information and preferences
  - use a credit card to purchase shopping cart contents
- The pet store will present a signon page to a customer who is not signed on whenever the customer:

- explicitly requests signon
- tries to update account information
- tries to purchase the contents of the shopping cart
- The signon page will allow registered customers to sign on and anonymous customers to register.
- The registration process automatically signs on the new customer.
- Customer registration requires: name, billing address, telephone number, email address, credit card number, credit card type, credit card expiration date.
- At registration, a customer may also indicate whether or not to define a “favorite” category, and whether or not to provide pet-related tips at the bottom of the screen.

### **User Signon And Customer Registration Module Design**

The key application design choice for this component is whether the application or the container manages security. The pet store manages user accounts in the application layer. It does not use any of the three login mechanisms specified by the J2EE platform (HTTP basic authentication, SSL authentication, or form-based login). Instead, it represents users and users' account information as entity beans. The pet store only requests user signon and registration when the user requests a “protected” page, which is a page that may be viewed only by signed-on users.

There are two kinds of users in an application: J2EE system users and application users. System users are created as users in the J2EE platform, using vendor-specific tools. Application users are represented and managed by application code.

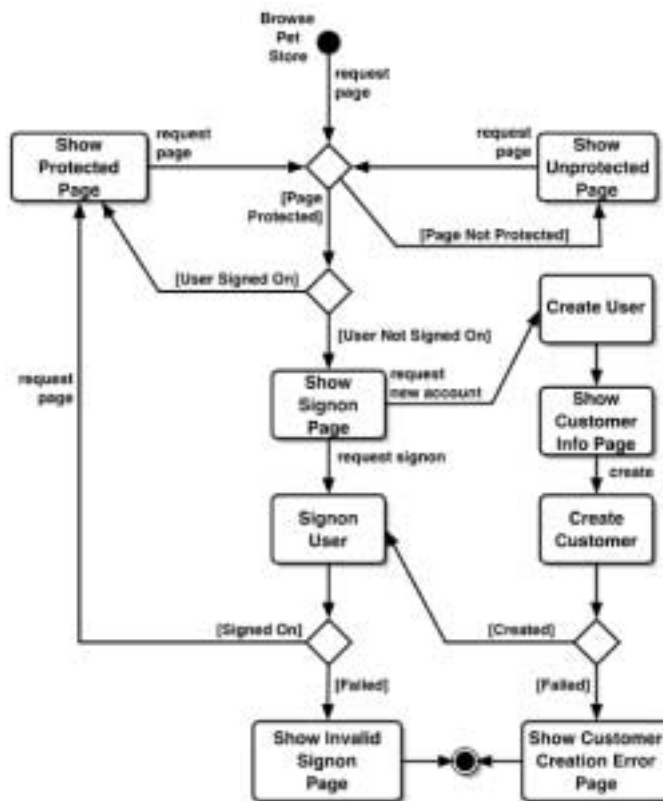
All users in the pet store are logged in as the same system user, so they all have the same system permissions, such as whether or not they can execute a certain method. Application permissions, such as who can create an order (registered users only), are modelled and controlled at the application level.

Casual users such as the shoppers in the pet store do not have to authenticate their real-life identity in any way. In such cases, it's usually preferable to manage users in the application layer instead of at the system layer. Users with special permissions, such as system or application administrators, are usually best represented by system users, using the J2EE login mechanisms mentioned above.

As a security measure, users with access to sensitive data and/or operations should always be created by a human system administrator. There is currently no

portable way to create J2EE system users programmatically. If you find you need to automatically create system users, consider writing non-portable code that leverages your platform's platform-specific user creation APIs to do so. Be sure to encapsulate any non-portable code behind an abstract interface, to make it "plug-gable" and easy to replace.

Figure 15 shows an activity diagram for the user signon and registration process. There are two kinds of pages in the pet store: unprotected and protected. Anyone may view unprotected pages, while only signed-on users may view protected pages. As shown in the diagram, unprotected pages are simply served directly to the customer.



**Figure 15** Activity Diagram for User Signon and Registration

When a customer requests a protected page, the pet store checks whether the customer is signed on as a user. If the customer is signed on, the requested page is

served. If the customer is not signed on, the pet store saves the request URL (for use after signon) and then redirects the request to the pet store signon form, shown in Figure 7.

The pet store signon page provides two options:

- **Signon as an existing user**—If the customer requests signon as an existing user, the pet store signs on the user, authenticating with the given user name and password. If user signon fails, an error page is displayed. The implementation of this process is explained in the section “User Signon and Customer Registration Module,” later in this document.
- **Signon as a new user**—If a customer requests creation of a new account, the pet store creates a new user with the given name and password, and then presents a customer information form. The customer fills out the form with information such as name, billing address, and credit card information, and then posts the form. If the pet store successfully creates the account, it immediately signs on the new user as described above. If account creation fails, an error page is displayed. The implementation of the user registration process appears in the section “User and Customer Creation Implementation,” below.

If user signon succeeds for an existing user, the pet store serves the originally requested URL, which was previously stored in session scope. Subsequent requests to protected pages are immediately directed to the requested page, because the user is already signed on.

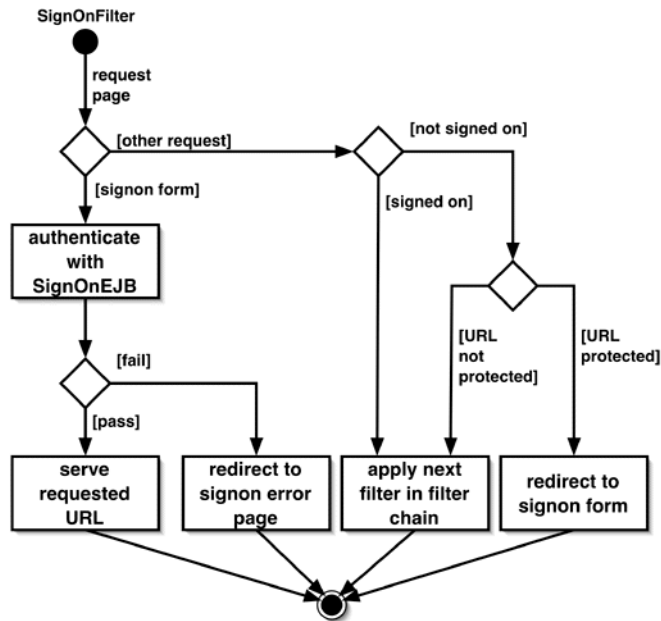
### **User Signon Implementation**

The user signon process is coordinated by the WAF component `SignOnFilter`, which detects requests for protected pages, signs on users, and forwards requests to appropriate URLs.

The `SignonFilter` component is an example of the design choice to use the Intercepting Filter pattern. An Intercepting Filter receives every request that is directed to the base component that it filters. It performs some operations, and then either passes on the request to its base component, or serves the request itself. An Intercepting Filter may also post-process a response created by the base component.

In the pet store Web site, certain URLs are “protected”, meaning that they may be viewed only by signed-on users. Intercepting Filter `SignonFilter` filters all HTTP requests under the application context root, protecting URLs and medi-

ating user signon. The `SignOnFilter` configuration file `signon-config.xml` defines both the signon page URL and which pages in the application are protected.



**Figure 16** Activity Diagram for `SignOnFilter` Component

An activity diagram for the `SignOnFilter` component appears in Figure 16. When the `SignOnFilter` receives a request, it first checks to see if the request responds to a POST of the existing user signon form. (The existing user signon form's name is by definition `j_signon_check`.) If the request is for the signon form, the filter authenticates the user name and password using stateless session bean `SignOnEJB`, which looks up the customer's `User` entity bean and validates the password. If signon succeeds, the originally-requested page (previously stored in session scope under the key `j_signon_original_URL`) is served. Signon may fail because the requested user does not exist, or because the password was incorrect. If signon fails, the request is redirected to the signon error page URL.

For all requests other than the signon page, if the user is signed on, the request is passed to the next filter in the servlet's filter chain. Subsequent filters in the chain then process the request, eventually producing a response. If the user is not

signed on and the URL is protected, the `SignOnFilter` saves the request URL for later use and redirects the request to the signon page, which the customer must then use to sign on. Finally, if the user is not signed on and the request is for an unprotected URL, the `SignOnFilter` passes the request to the next filter in the servlet filter chain.

The `SignOnFilter` allows the developer to use any HTML form called `j_signon_form` as an application's existing user signon form. A signon form may even appear on more than one page in the same application. The `SignOnFilter` may be used even in J2EE applications that don't use the WAF. The only requirements are that the `SignOnFilter` have access to the `SignOnEJB` session bean, and that the `SignOnEJB` have access to the `User` entity bean.

`SignOnFilter` only signs on existing users. Registration of new users is handled with WAF action classes, as described in the next section.

### **User and Customer Creation Implementation**

The pet store signon page shown in Figure 7 contains two HTML forms: `j_signon_check` to sign on existing users (described previously), and `newcustomer` for new customer registrations. The `newcustomer` form posts a requested new user name and the password, typed twice to ensure accuracy.

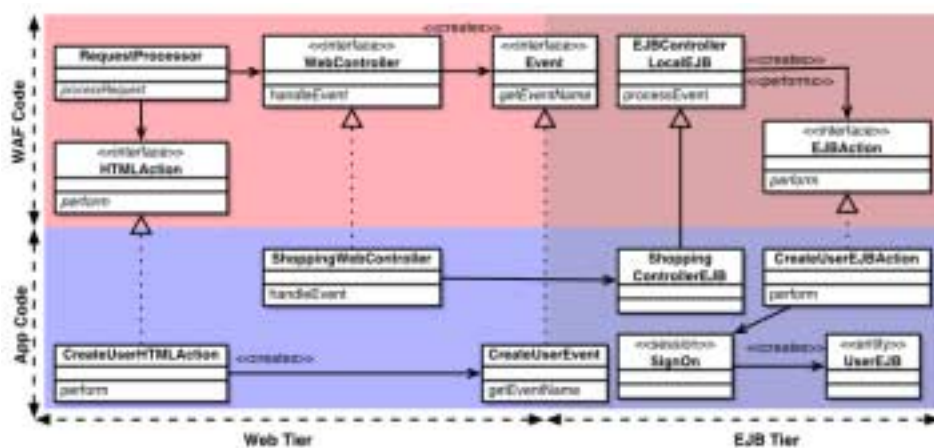
A pet store customer is represented by several entity beans. A `User` bean represents the customer's login name and password. Other entity beans represent the customer, profile, accounts, contact info, address, and credit cards. The `User` entity bean, which corresponds to a logged-in user, is deliberately separated from the `Customer` and associated entity beans to avoid coupling user authentication data (potentially used by multiple applications) with application-specific data (such as a pet store customer).

### **User Creation**

The pet store uses the WAF to create a new user from the `newcustomer` form in `signon.jsp`. User creation provides a good example of how the WAF can be extended to provide application-specific functionality.

Figure 17 shows the key WAF classes and interfaces involved in creating a new pet store user. Based on the request URL, the `RequestProcessor` creates and executes an HTML action (HTMLAction subclass `CreateUserHTMLAction`) that performs any business logic specific to the Web tier. Because the business logic requires activity in the EJB tier, `CreateUserHTMLAction` returns an `Event` (`CreateUserEvent`), which the `RequestProcessor` passes to the EJB controller (EJB-

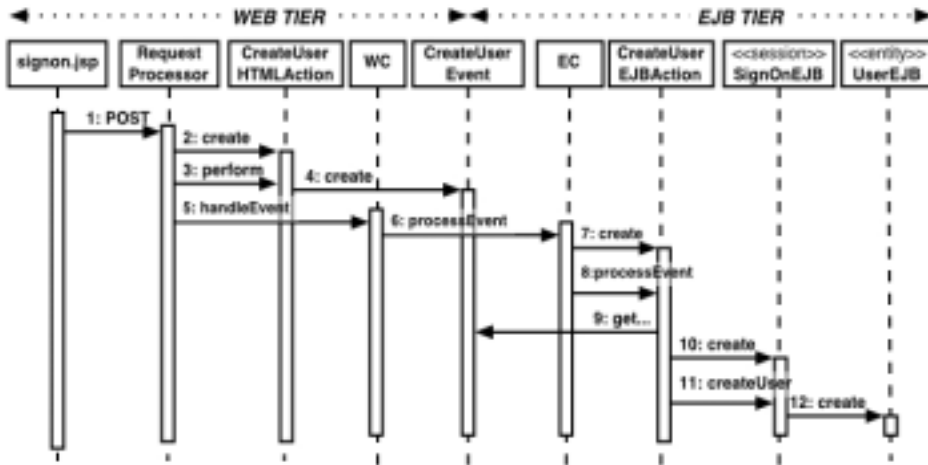
ControllerLocalEJB) by way of the Web controller (interface WebController, implemented by class DefaultWebController). An Event represents a command to be executed by the EJB controller (stateful session bean EJBControllerLocalEJB extended by class ShoppingControllerEJB). The EJB controller builds and evaluates an EJB action (EJBAction subclass CreateUserEJBAction) that creates the User entity. Note in the diagram that control flow is handled by the framework (the classes and interfaces across the top of the diagram), while application-specific code executes in developer-defined subclasses.



**Figure 17** Classes And Interfaces Involved In User Creation

An object interaction diagram of the user creation process appears in Figure 18. The following steps describe the user creation process in detail (see the corresponding numbers in the diagram).

1. The ACTION for the new customer form is `createuser.do`. A servlet mapping in the pet store Web deployment descriptor `web.xml` maps all requests matching `*.do` to the `MainServlet`. When the browser POSTs form `newcustomer` to the URL `createuser.do`, `MainServlet` passes the request to the `RequestProcessor`.



**Figure 18** User Creation Object Interaction Diagram

- The excerpt from the pet store URL mappings file mappings.xml shown in Code Example 4 says that request URL createuser.do is processed by CreateUserHTMLAction, and that the next screen to display is create\_customer.screen.

```
<url-mapping url="createuser.do" screen="create_customer.screen">
  <web-action-class>
    com.sun.j2ee.blueprints.petstore.controller.web.actions.CreateUser
    HTMLAction
  </web-action-class>
</url-mapping>
```

**Code Example 4** Mapping URL createuser.do to HTMLAction and Next Screen

The request processor looks up URL createuser.do in the request map and creates an instance of CreateUserHTMLAction.

- The request processor calls CreateUserHTMLAction.perform on the instance it created.
- CreateUserHTMLAction.perform creates CreateUserEvent, initializing it with the user name and password, and returns the event to the request processor. The

event encapsulates a command to the EJB Client Controller (EC) in the EJB tier, instructing it to create a new user. The RequestProcessor looks up the event to EJB action mapping in the mappings.xml file and calls the setEJBActionClassName method on the CreateUserEvent. Code example 5 shows how the event CreateUserEvent maps to an EJBAction class CreateUserEJBAction.

```
<event-mapping>
  <event-class>
com.sun.j2ee.blueprints.petstore.controller.events.CreateUserEvent
  </event-class>
  <ejb-action-class>
com.sun.j2ee.blueprints.petstore.controller.ejb.actions.CreateUserEJBAction
  </ejb-action-class>
</event-mapping>
```

**Code Example 5** Mapping CreateUserEvent event to CreateUserEJBAction

5. The request processor passes the event to the Web Client Controller (shown as “WC” in the diagram) method handleEvent. The WC is implemented in the pet store by class DefaultWebController.
6. The WC passes the event in the Web tier to the EJB Client Controller (“EC” in the diagram) in the EJB tier, by calling the EC’s method processEvent. The EC is the EJB controller shown in Figure 14, and is implemented in the pet store Web site by class ShoppingControllerEJB.
7. The EC creates an instance of CreateUserEJBAction, which actually creates the user. The EC uses Event method getEJBActionClassName to determine which class to instantiate. CreateUserEJBAction. This level of indirection provides deployment-time selection of business logic by choosing an EJBAction class for an Event.
8. The EC calls the CreateUserEJBAction.processEvent, passing it the CreateUserEvent.
9. The action calls the event get methods getUsername and getPassword to get the data it needs to create the new user.
10. The action creates a reference to a SignOnEJB, which is a stateless session bean that is used to create and authenticate users.
11. The action creates a User entity bean, which persistently stores the user name

and password.

Once a user is created, the next step is to create the customer entities.

### Customer Creation

Code Example 4 indicates that after creating a user, the WAF serves screen `create_customer.screen`. The customer registration form appears in Figure 8. The pet store converts data submitted in this form into a collection of interlinked entity beans in the EJB tier. A class diagram of the classes involved in tracking user identity appear in Figure 17. The Customer is linked to its corresponding User by way of its `userId` field, which matches the User field `userName`.

The customer fills out the registration form and clicks *Submit*. The HTML form POSTs data to the URL `customer.do`. When the application creates a customer, the WAF executes exactly the same sequence of actions as when it creates a user. The only difference is that the application-specific classes create customer information instead of creating a user. Here are the steps performed by the WAF when the application is creating a customer:

1. Based on `mappings.xml`, the `RequestProcessor` maps `customer.do` to an instance of `CustomerHTMLAction` and calls the action's `perform` method.
2. `CustomerHTMLAction.perform` constructs and returns a `CustomerEvent`, which contains the data from the form.
3. The `RequestProcessor` sends the `CustomerEvent` to the EJB controller by way of the Web controller.
4. Based on the event to action class name mapping for the `CustomerEvent`, the EJB controller creates an instance of `CustomerEJBAction` and passes the `CustomerEvent` to its `handleEvent` method.
5. Using the data encoded in the `CustomerEvent`, `CustomerEJBAction.handleEvent` creates and initializes a new Customer entity bean and all of its associated beans (shown in Figure 21).
6. Method `CustomerHTMLAction.doEnd`, which is called after the `EJBAction` executes, signs the user on to the system if it was created successfully.
7. The screen flow manager forwards the request to `customer.screen`, which summarizes customer data for the user.

### 1.2.5.3 Asynchronous Messaging Module

The pet store AsyncSender stateless session bean formats a purchase order as an XML document and transmits it to the Order Processing Center (OPC) for fulfillment.

#### Asynchronous Messaging Module Requirements

The communication between the pet store and the OPC must meet the following requirements:

- Communication must be asynchronous. Because the order fulfillment process may take a long while, the pet store must be able to create and order and continue, not waiting for the order to complete.
- Message delivery must be reliable. When the pet store sends a message, the OPC must receive and process the message exactly once.
- If the OPC is not available when the message is sent, the message must be stored and delivered when the OPC becomes available.
- The interface between the OPC and the pet store must be flexible. It should be easy to change the implementation of either the pet store or the OPC.
- The interface to the OPC should be easily extendable to clients other than the pet store, including clients not written in the Java language.
- Each order transmitted by the pet store must include a globally-unique identifier.

#### Asynchronous Messaging Module Design

Several features of the J2EE platform make the pet store to OPC communication requirements easy to meet. The following design choices directly address the requirements listed above:

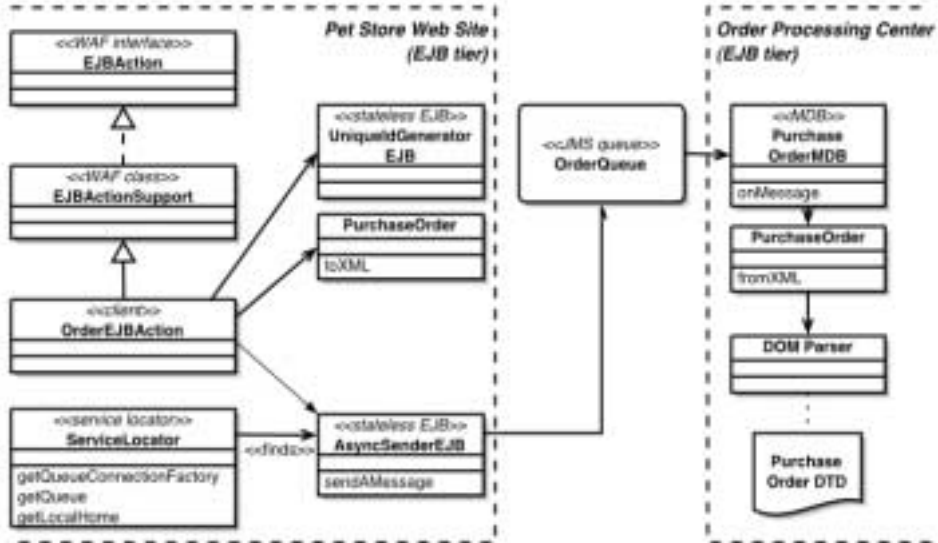
- The pet store and OPC use JMS to communicate, because it provides asynchronous, reliable, once-only message delivery.
- JMS providers offer store-and-forward message delivery that ensures that messages are delivered even if the message consumer is temporarily unavailable.
- The pet store uses a JMS queue (rather than a topic) to send the purchase order messages because such communication is point-to-point.

- Purchase orders are represented as XML messages, decoupling the pet store and OPC interfaces from one another.
- Because XML is technology-independent, clients other than the pet store, including clients not written in the Java language, could create purchase orders for the OPC. Using XML to represent the purchase order is a first step towards providing a Web service interface to the OPC.

### **Asynchronous Messaging Module Implementation**

The WAF initiates the process of creating a purchase order for transmission to the OPC, using the Web actions, EJB events, and EJB actions. When a pet store user first requests purchase of shopping cart contents, the pet store signon filter (described in the section “User Signon and Customer Registration Module,” above) ensures that the user is signed on, and the template engine assembles and serves screen `enter_order_information.screen`. The JSP page containing the form for this screen, `enter_order_information.jsp`, posts its contents to the pet store URL `order.do`. Following the configuration in `mappings.xml`, the WAF creates an `OrderHTMLAction`, which assembles the order information into an `OrderEvent`, which is sent to the EJB tier for processing by an `OrderEJBAction`.

A diagram of the classes involved in sending the purchase order appears in Figure 19. The `OrderEJBAction` performs the EJB-tier business function of building and transmitting a `PurchaseOrder` (PO) object (represented as an XML message) to the OPC by way of JMS. As Figure 19 shows, the action class creates a new purchase order. It uses `UniqueGeneratorEJB` to create an order ID, which is a globally-unique identifier for the purchase order. It also sets the purchase order’s shipping and billing address and credit card number, based on data that it received from the `OrderEvent`. The `OrderEJBAction` then iterates over the collection of items in the shopping cart, adding the items to the purchase order.



**Figure 19** Asynchronous Transmission of Purchase Order from Pet Store to OPC

Finally, the `OrderEJBAction` sends an XML representation of the purchase order to the OPC. The `ServiceLocator` is used to lookup the `AsyncSender EJB`. The `AsyncSender EJB` is a stateless session bean which handles all the details of looking up the messaging service and sending the purchase order XML document to the queue, `OrderQueue`. Note that this delivery is asynchronous, and once the `AsyncSender EJB` sends the purchase order to the JMS queue called `OrderQueue`, control returns to the user. At some later time, a message-driven bean (MDB) in the OPC `PurchaseOrderMDB` asynchronously receives the purchase order message, while the pet store continues serving other requests. The pet store maintains no history of the orders it has created. Order information and order tracking is managed by the OPC.

The `PurchaseOrder (PO)` class that the pet store uses to represent the purchase order is a conventional class written in the Java language that maintains information about purchase order: the order ID, the user ID, the billing and shipping addresses, the line items, and so on. It also has a method `toXML` that returns an XML representation of the purchase order as a string. Method `toXML` builds an in-memory representation (a DOM tree) of the `PurchaseOrder` document, and serial-

izes that representation to a `String`. The pet store sends the XML string as the payload of a JMS text message, using `AsyncSender` as described above.

Now, just to briefly illustrate the message passing between the pet store application and the back end order processing center application (OPC), lets take a quick look at the OPC application functionality when a new purchase order arrives. Since the communication between the pet store and the OPC is asynchronous, at some time later, the JMS provider delivers the purchase order XML document from the `OrderQueue` to the OPC application. When the OPC `PurchaseOrderMDB` receives the message, it creates and initializes a new `PurchaseOrder` instance with data from the received XML string, using the static method `PurchaseOrder.fromXML`. The method `fromXML` will validate the purchase order document and then return a new `PurchaseOrder` object initialized with data from the XML document sent from the pet store.

When method `fromXML` parses the message, its DOM parser validates the received XML against the document type definition (DTD) `PurchaseOrder.dtd`. It uses a validating DOM parser to construct a DOM tree from the received string, and then initializes the purchase order object's internal fields from the values in the DOM tree. Using validation means that the parser, not the application code, checks the integrity of the data in the purchase order XML document. By using a validating parser, a developer can simply describe the data integrity rules in the DTD and let the parser check the document, instead of writing (and maintaining) a large quantity of tedious value-checking code. Moreover, if the format of the XML document changes, the validation rules can be changed in the DTD instead of in application code.

When the OPC application receives a new purchase order, it begins the order fulfillment process that is done by the order processing center (OPC) application and the supplier application. This completes the submitting of an order for the pet-store application since all the order fulfillment processing occurs in the background.

#### **1.2.5.4 Catalog Module**

The catalog module is the interface to the catalog of items for sale in the pet store.

##### **Catalog Module Requirements**

The catalog module must meet the following requirements:

- Each individual item shipped by the pet store enterprise must have a unique

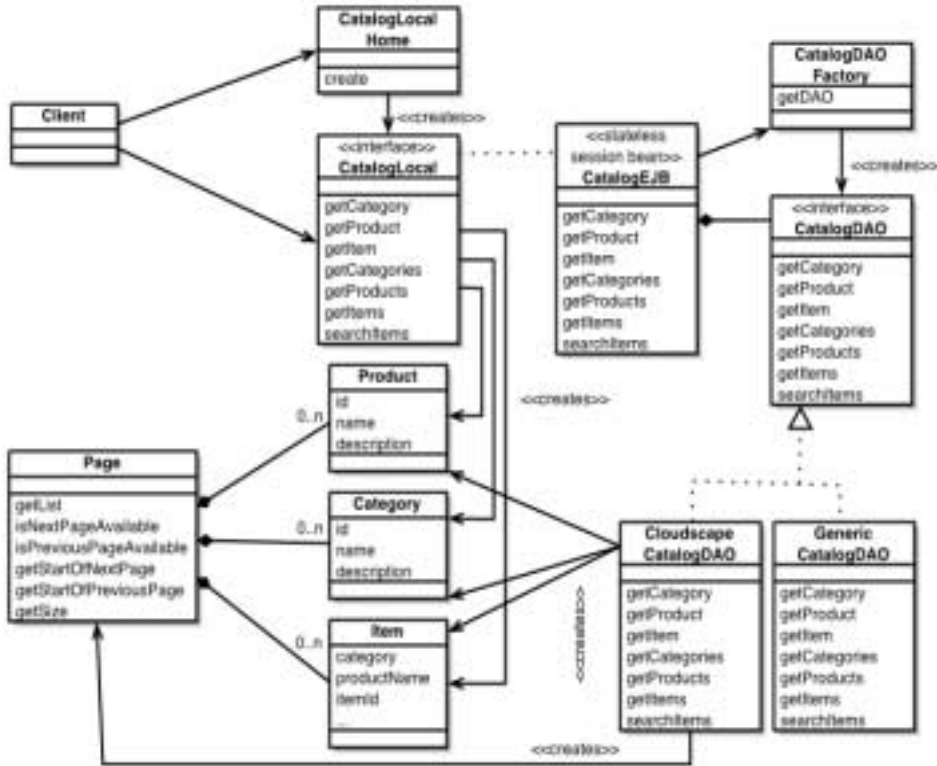
identifier plus descriptive information.

- A product is a group of catalog items, with an identifier, a name, and a description.
- A category is a group of related products, with an id, a name, and a description.
- The catalog module must also provide read-only access to lists of categories, products, and items by “page”. A page is a sublist of a specific length, starting with a specific entry, of the fully-ordered list of categories, products, or items.
- The catalog component must be usable for multiple client types.
- It must be possible to add categories, products, and items to the database out programming.
- Because the catalog is the most highly-used component in the application, it must be as efficient and scalable as possible.
- The storage mechanism used for the catalog contents should be easy to change.
- It should be easy for the catalog module to be converted to a remote component, if desired.

### **Catalog Module Design**

Figure 20 shows a structure diagram of the catalog design. A client program accesses the catalog to retrieve information about catalog entries (categories, products, and items), either individually or page-by-page. The catalog contains multiple categories, which contain multiple products, which may contain multiple items. A page may contain any number of either categories, products, or items, but all entries

on the page must be of the same type. The catalog gets its entries from an external read-only data store.



**Figure 20** Catalog Structure Diagram

The key design choice for this module was to use a stateless session bean to access back-end catalog data through a data access object (DAO), instead of representing and accessing data as entity beans. In the Web site user views, catalog data are displayed in much the same way as they are stored—as tables. In addition, catalog data is read-only in the pet store Web site. In situations when relational data are being accessed and used in tabular form, and especially when access is read-only, it's preferable to access the data relationally, using a DAO to encapsulate the details of the data access mechanism. In such situations, using entity beans incurs a performance penalty while providing little or no additional value.

The pet store Web site leverages J2EE technologies and design patterns to meet the requirements for the catalog. The specific design choices made for this design are:

- **The catalog is a stateless session bean.** If the catalog were implemented as a Web-tier component like a servlet, it would be limited to Web-only clients. If the catalog were written as a conventional class, it might have to manage connection pooling and concurrent access to achieve adequate performance and scalability. Choosing a stateless session bean provides catalog access to any type of enterprise beans client, while meeting the resource sharing and scalability goals of the component. The component is stateless because it requires the highest possible performance and scalability.
- **Catalog entries are modelled as conventional classes.** In this application, catalog entries provide an excellent example of when not to use enterprise beans. Catalog entries are simple, tabular data with little or no behavior. They do not need the services enterprise beans provide. For example, because they are read-only, they don't require concurrent update management or transactions. Modelling catalog entries as entity beans would create unnecessary overhead. Modelling them as conventional classes meets all of their requirements in the simplest, most efficient way.
- **Catalog entries are serializable.** If the catalog at some future time is changed to be a remote component, the entries can be passed through the remote interface as Transfer Objects, and cached on the client side.
- **The catalog uses the Data Access Object (DAO) pattern to read the catalog data.** The DAO pattern provides access to data while encapsulating and hiding the data access mechanism. Because catalog entries are not entity beans, their persistence must be managed in code. The catalog module DAO uses JDBC to read catalog entries directly from the EIS tier. The DAO pattern hides the differences in SQL implementations between vendors. The pattern also makes it easy to change how the catalog accesses its entries.
- **Categories, products, and items are stored in a database.** A database is most flexible and highest-performance storage mechanism for data like the catalog. Using a database, the entire catalog can be loaded with database administration tools, and it can be modified using either SQL or a custom database client.

### Catalog Module Implementation

The implementation of the catalog follows the design closely. Referring again to Figure 20, the `CatalogLocal` interface is implemented by the `CatalogEJB` stateless

session bean, which is created by `CatalogLocalHome`. The `CatalogLocal` methods return instances of `Category`, `Product`, and `Item`, either individually or grouped by page. Class `Page` groups multiple instances of either `Category`, `Product`, or `Item` objects, but does not mix them. Clients using the `Page` class must cast the elements in the `Page` to the appropriate class. `CatalogLocal` also has a method, `searchItems`, which performs a search of `Item` objects based on supplied search criteria.

The `CatalogEJB` gets its data from a `CatalogDAO`. The `CatalogDAO` is created by a `CatalogDAOFactory`, making the DAO “pluggable”. New sources of catalog data can be accommodated simply by writing a new class that implements `CatalogDAO`, configuring the factory to create instances of the new class, and redeploying the application. The existing `CatalogDAO` implementation, `CloudscapeCatalogDAO`, retrieves the data from a relational database using JDBC.

In the sample application, the JSP pages that display the categories, products, and items play the role of the client, requesting pages of categories, products, and items, or displaying the details of any of these. When the user request the “previous” or “next” page of entries, a tag in the JSP page requests the next page from the catalog and displays it.

### **1.2.5.5 CustomerModule**

The customer module tracks information about pet store customers, and associates these customers with the user information that they used to signon to the system.

#### **Customer Module Requirements**

The customer module has the following requirements:

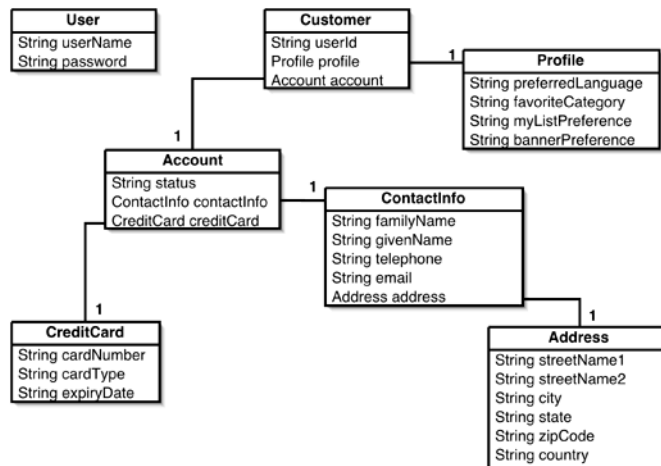
- The customer module must track a customer’s customization profile (such as preferred language, favorite category, etc.), contact information, and credit card information.
- The module must be accessible to both HTTP and non-HTTP clients.
- Customer data must be persistent and identifiable by the customer’s system user id.
- Customer data must be usable from multiple application components.

## Customer Module Design

There were three options for modelling the customer module components: local entity beans, conventional Java classes, and remote entity beans. The customer module is modelled as a network of local entity beans with fine-grained APIs. The design choice to use local entity beans to model fine-grained business entities comes from the more general decision to use a local architecture. Local entity beans were chosen over conventional Java classes because they combine the benefits of enterprise beans (CMP, CMR, transactions, etc.) with performance approaching (or even surpassing) that of non-managed business classes. Previous versions of the pet store customer module defined a coarse-grained remote interface that returned transfer objects for efficient fine-grained access. Switching to a local architecture made a fine-grained design practical, and eliminated the transfer object layer, thus simplifying the design.

Keep in mind that the combination of coarse-grained, remote component interfaces that return transfer objects is still appropriate in distributed architectures. For more on transfer objects, see the reference to the Transfer Object pattern in “Resources and References,” at the end of this document.

## Customer Module Implementation



**Figure 21** Local Entity Beans That Track Customer Information

Figure 21 is a structure diagram of the local entity beans that track customer data. (To simplify the diagram, the combination of local component and home interfaces and implementation class are shown here as a single class. For example, the Customer entity bean in Figure 21 is actually composed of interfaces CustomerLocal, CustomerLocalHome and class CustomerEJB.)

Notice that there is no explicit relationship between the User and Customer classes. This separation was a deliberate design decision that decouples the user signon component (discussed in the section “User Signon and Customer Registration Module,” above) from the customer component. Because the two modules are decoupled, other applications can use the signon module without also having to use the customer module.

Local entity beans are the clear choice for modelling customer data in this application. Very little code was necessary to create these beans, because the EJB container not only manages the persistence of attributes and relationships, it even writes the property accessors. For example, the definition of the Customer’s property accessors in class CustomerEJB appears in Code Example 6.

```
// getters and setters for CMP fields
//=====
public abstract String getUserId();
public abstract void setUserId(String userId);

// CMR Fields
public abstract AccountLocal getAccount();
public abstract void setAccount(AccountLocal account);

public abstract ProfileLocal getProfile();
public abstract void setProfile(ProfileLocal profile);
```

**Code Example 6**      Definition of Customer’s Abstract Property Accessors

The property accessors (“getters” and “setters”) for the `userId` property are defined by the developer as abstract, and the EJB container automatically creates the implementations for the methods. In EJB 2.0 CMP, the internal state of an entity bean is no longer kept in fields; instead, it is available only through these accessors. If the Customer bean wants to do something with its own `userId`, it calls its own `getUserId` method, instead of using an internal field.

The automatic container-managed relationship feature of EJB 2.0 CMP greatly simplifies managing relationships between entity beans. Looking again at Code Example 6, notice the methods `getAccount` and `setAccount`. These two methods look like property accessors for a property called `account`, but actually they maintain the relationship between a `Customer` and the customer's `Account`. This mechanism allows developer code to create, delete, and modify persistent relationships between entity beans by calling methods implemented by the container.

### **1.2.5.6 Shopping Cart Module**

The shopping card module tracks customer purchases during a customer shopping session

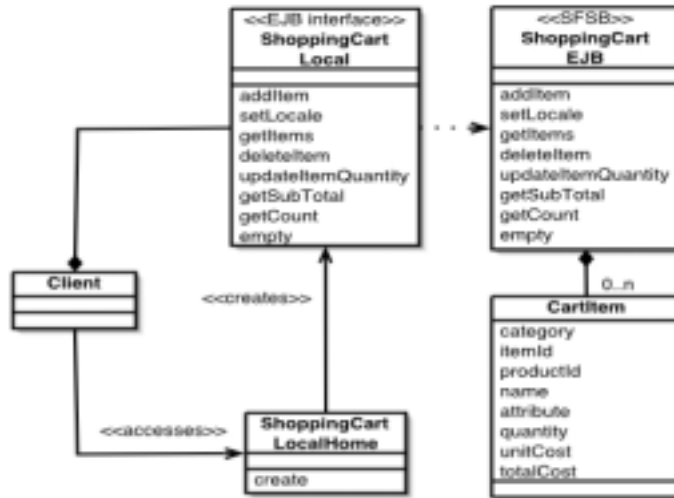
#### **Shopping Cart Module Requirements**

The shopping cart has the following requirements:

- It must track multiple items that the user has selected.
- An item in the cart is identified by its item ID, and includes the item's product ID, its category, its name, and its unit cost.
- There must be a nonnegative quantity associated with each item.
- The shopping cart's clients must be able to add, remove, and iterate items, change an item's quantity, and entirely empty the cart.

#### **Shopping Cart Module Design**

The shopping cart contents clearly belong in session scope, because they are specific to a user, and they continue to exist across multiple requests. The key design decision on the shopping cart module was where to store the session state containing the cart. Because this design already uses an EJB tier, the obvious choice was to use a stateful session bean. Stateful session beans provide automatic lifecycle management and thread safety for session state. Furthermore, placing session state in the EJB tier makes the state accessible to all enterprise beans clients, instead of only to Web clients. See "Web Tier State Recommendations" in the Web tier chapter of the BluePrints book for more on how to decide where to store state.



**Figure 22** Structure Diagram of Shopping Cart Module

A structure diagram of the shopping cart module appears in Figure 22. The client (currently, this client is the Web application itself) uses the `ShoppingCartLocalHome` to create a `ShoppingCartLocal` bean. The enterprise bean class for this interface is `ShoppingCartLocalEJB`, which contains methods that perform all of the operations that the requirements specify. The `ShoppingCart` maintains a private collection of `CartItem` objects, each of which is a conventional Java transfer object class that simply stores the attributes of an item, and the quantity of that item the user wishes to order.

### Shopping Cart Module Implementation

The shopping cart module implementation is a straightforward stateful session bean, which maintains a collection of `CartItem` objects in a private `HashMap`. Session bean methods can be used to add, remove, get, and update items, as well as to empty the cart.

EJB-tier components may access the shopping cart module directly. The Web tier maintains a reference to the EJB-tier controller in an `HttpSession` attribute. The EJB-tier controller's method `getShoppingClientFacade` returns a reference to a `ShoppingClientFacadeLocal` interface. The shopping client facade's `getShoppingCart` method returns the current session's shopping cart.

This concludes the discussion of the pet store Web site application module designs.

## 1.2.6 Pet Store Web Site Application Components

This section is a simple annotated list of the components and files in the pet store Web site. It is just a quick reference to all the components and files in the petstore. The pet store comprises several types of application-specific code and data:

- **enterprise beans**—that represent business data and perform business logic
- **JSP pages**—that define an application view template (`template.jsp`) and the contents of the areas of the template. Also included are various graphics files used by the JSP pages.
- **XML files**—that define screens, control screen flow, bind request URLs to HTML actions, define user signon, and define the sample database contents; also standard J2EE deployment descriptors
- **servlet filters**—that coordinate user signon and enforce response encoding
- **an asynchronous sender component**—that constructs and transmits XML messages representing orders to the Order Processing Center
- **a populate program**—that populates the sample database

### 1.2.6.1 Pet Store Enterprise Beans

The pet store uses session beans to implement business processes, and entity beans to represent and manipulate business data. All beans in the pet store provide only a local interface to their clients. The pet store requires product that complies with J2EE 1.3. It also will not run on J2EE products that do not provide local interface access from the Web tier, because some Web tier components use enterprise bean local interfaces.

#### Session Beans

The session beans used by the pet store are:

- **AsyncSenderEJB**—(Stateless) converts shopping cart contents and customer data into an XML message representing an order, and sends the message to the Order Processing Center. For more on this bean, see the section “Asynchro-

nous Sender,” above.

- **CatalogEJB**—(Stateless) provides a programmatic interface to the catalog
- **ShoppingCartEJB**—(Stateful) maintains the contents of an individual user’s virtual shopping cart
- **ShoppingControllerEJB**—(Stateful) provides access to the Shopping Client Facade; extends the WAF EJB controller (`EJBControllerLocalEJB`)
- **ShoppingClientFacadeEJB**—(Stateful) caches references and provides unified access to customer, shopping cart, and user ID
- **SignOnEJB**—(Stateless) creates and authenticates system users
- **UniqueIdGeneratorEJB**—(Stateless) creates globally unique object identifiers

### Entity Beans

The pet store uses the following entity beans. All beans use container-managed persistence.

- **CustomerEJB**—tracks customer ID (primary key), account, and profile
- **AccountEJB**—tracks account status, credit card, and contact info
- **ProfileEJB**—tracks preferred language, category, list preference, and banner preference
- **ContactInfoEJB**—tracks family and given name, telephone, email, and address
- **CreditCardEJB**—tracks card number, card type, and expiration date
- **AddressEJB**—tracks two lines of street address, state, zip code, and country
- **UserEJB**—represents a user who may sign on to the system; tracks a user name and password
- **CounterEJB**—represents a counter with a specific prefix; used only by `UniqueIdGeneratorEJB` to manage series of unique numbers

All of the entity beans in the pet store, with the exception of `CounterEJB`, track customer information. A diagram of the classes involved in tracking customers’ identity in the pet store appears in Figure 21.

### 1.2.6.2 Pet Store JSP pages

The pet store defines the JSP pages shown in Table 1. Most pages have a corresponding page localized for Japanese.

**Table 1** JSP Pages Defined By The Pet Store

JSP Page Name	Description
advice_banner.jsp	Displays tips at bottom of screen, if tips are turned on
banner.jsp	Displays banner at top of screen, if banner is turned on
cart.jsp	Displays shopping cart contents
cart_empty_order_error.jsp	Error page when user attempts to purchase contents of empty cart
create_customer.jsp	Form to create a customer
customer.jsp	Displays customer information
duplicate_account.jsp	Error message reporting that user tried to create an account with an existing name
edit_customer.jsp	Form to edit customer information
footer.jsp	Displays message in page footer
general_error.jsp	Displays miscellaneous errors not covered by other error pages
index.jsp	Front page of sample application
item.jsp	Displays catalog item details
items.jsp	Displays list of items from a catalog search or browse
main.jsp	Main pet store opening screen with large parrot graphic and links to categories
mylist.jsp	Displays pet favorites of user
order_completed.jsp	Acknowledges receipt of order from customer and reports the order number
product.jsp	Displays a list of products for a particular category

**Table 1** JSP Pages Defined By The Pet Store

JSP Page Name	Description
advice_banner.jsp	Displays tips at bottom of screen, if tips are turned on
search.jsp	Search form
sidebar.jsp	Side bar menu
signoff.jsp	Acknowledges that user has signed off
signon.jsp	Allows system signon as an existing user or as a new account
signon_failed.jsp	Reports that signon failed and provides a reason
template.jsp	Defines page layout for entire application

### 1.2.6.3 Pet Store Servlet Filters

The pet store defines two servlet filters:

`SignOnFilter`—Coordinates user sign on and customer registration. This filter is described in detail in the section “User Signon and Customer Registration Module,” above.

`EncodingFilter`—Ensures that every page the pet store serves to browsers has the appropriate encoding.

### 1.2.6.4 Pet Store XML Files

Several XML files support pet store operation. Several are deployment descriptors, as described in the *Packaging and Deployment* chapter of the BluePrints book:

- **application.xml**—the standard J2EE deployment descriptor for the pet store Web site J2EE application
- **web.xml**—the standard J2EE deployment descriptor for the pet store Web-tier components
- **ejb-jar.xml**—the standard J2EE deployment descriptors for the pet store EJB-tier components. There are several of these, one for each enterprise beans deployment unit.
- **sun-j2ee-ri.xml**—the vendor-specific deployment descriptor, containing de-

ployment information specific to the J2EE reference implementation. Each platform implementation needs such a descriptor to contain deployment information specific to that implementation. See your J2EE product documentation for details.

Other XML files in the pet store are specific to the pet store application:

- **Populate-UTF8.xml**—contains the data used to populate the sample database
- **PopulateSQL.xml**—contains the Cloudscape and Oracle SQL statements to create the sample database. Cloudscape is the relational database shipped with the J2EE reference implementation.
- **CatalogDAOsQL.xml**—contains the Cloudscape and Oracle SQL statements that are used by the CatalogDAO to access the data in the sample database.
- **mappings.xml**—defines both the pet store request map and pet store screen flow map, as described in the section “Control Module Implementation,” above.
- **screendefinitions\_en\_US.xml**—defines screens for application localized in US English
- **screendefinitions\_ja\_JP.xml**—defines screens for the application localized in Japanese
- **signon-config.xml**—configures the SignOnFilter

#### 1.2.6.5 Asynchronous Sender

The pet store uses the AsyncSender component to create orders in the Order Processing Center (OPC). The AsyncSender component is a stateless session bean that transforms the pet store shopping cart contents and customer and billing information into an XML document, and sends the document to the OPC as a JMS message. The OPC application begins the fulfillment process on receipt of the message.

#### 1.2.6.6 Populate Servlet

The PopulateServlet creates the database for the sample application. It is designed to be portable across J2EE platform implementations. It creates persistence schema using DDL from PopulateSQL.xml, and then creates instances of entity beans using data from the file Populate-UTF8.xml. The populate servlet executes the first time a

user accesses the pet store after installation. It may also be executed forcibly from the pet store main page.

### 1.3 Resources and References

- The “BluePrints book” referenced in this chapter is:  
*Designing Enterprise Applications with the Java™ 2, Enterprise Edition, 2/e*  
Inderjeet Singh, Beth Stearns, Mark Johnson, and the Enterprise Team  
ISBN: 0201787903
- The J2EE patterns book referenced in this chapter is:  
*Core J2EE™ Patterns*  
Deepak Alur, John Crupi, Dan Malks  
ISBN: 0130648841
- J2EE Design Pattern references:
  - Data Access Object—  
<http://developer.java.sun.com/developer/restricted/patterns/DataAccessObject.html>
  - Front Controller—  
<http://developer.java.sun.com/developer/restricted/patterns/FrontController.html>
  - Intercepting Filter (also known as Decorating Filter)—  
<http://developer.java.sun.com/developer/restricted/patterns/DecoratingFilter.html>
  - Transfer Object (previously known as Value Object)—  
<http://developer.java.sun.com/developer/restricted/patterns/ValueObject.html>
- For more on the tradeoffs involved in using a Web-tier application framework, see “Web-Tier Application Framework Design” in the Web Tier chapter of BluePrints book.
- For more on servlet filters, see “Java Servlet Technology” in the J2EE™ Tutorial:  
<http://java.sun.com/j2ee/tutorial/doc/Servlets.html>
- To find out about Front Controllers, see “Web-Tier MVC Controller Design” in the BluePrints book. See also the Front Controller design pattern in *Core*

*J2EE Patterns.*

- Details about the template mechanism of the pet store Web site are explained in the section “Templating” in the Web Tier chapter of the BluePrints book.