

IBM WebSphere Commerce



Payments Cassette Kit Programming Guide

Version 5.5

IBM WebSphere Commerce



Payments Cassette Kit Programming Guide

Version 5.5

Note

Before using this information and the product it supports, be sure to read the general information under “Notices”, on page 209.

Seventh Edition (July 2003)

© Copyright International Business Machines Corporation 2000, 2003. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Welcome!	v
Conventions used in this book	vi
Additional information	vii
Using the online help	vii
Locating the printable documentation	vii
Viewing the WebSphere Commerce Web site for product information	vii
Other WebSphere Commerce Payments documents and Web sites	vii
What's new for release 5.5	viii

Chapter 1. Introducing WebSphere Commerce Payments	1
WebSphere Commerce Payments features and goals	1
WebSphere Commerce Payments architecture	2

Chapter 2. Understanding the WebSphere Commerce Payments framework.	7
--	----------

Framework object model	7
Administrative object model	8
Financial objects and their states	11
Exported Data Model: The user's view of WebSphere Commerce Payments objects.	22
Framework commands	24
Command processing overview	25
Protecting sensitive data	29
Cassettes and command processing	31
Administrative API commands	39
Administrative command sequence diagrams	41
Payment API commands	61
Payment command sequence diagrams	67
Query command processing	90
Responsibilities and services.	91
Access control	91
Order and batch caching	91
Threading	92
Synchronization	94
Parameter validation	96
Background and timed operations.	99
Receiving protocol messages from the outside world	102
Database access	102
Event notification	105
Map your cassette AVS codes to the WebSphere Commerce Payments common AVS result codes.	105
Asynchronous Auto Approve	107
Account settings related to AcceptPayment and ReceivePayment	109
Configurable approval expiration.	111
Purchasing card support.	112
Error logging	115
Return code messages	116
Debug tracing	119

User interface support	123
Payment Server Presentation Language.	123
User's guide to PSPL	128
PSPL reference	133
Framework Javadoc	154
Cassette view of framework classes	154

Chapter 3. Designing your cassette **157**

Design activities	157
Name your cassette	157
Consider internationalization	158
Design your extensions to the framework object model.	160
Map the WebSphere Commerce Payments API to your payment protocol	164
Design your commit points.	165
Consider restart implications	165
Design ComPoints and ETillConnection	167
Design financial state transitions	167
Create scenario diagrams	168
Write your cassette documentation	168

Chapter 4. Writing your cassette **171**

Installation and uninstallation considerations and steps	173
Installing and configuring your cassette	173
Step 1: Create the directory structure	173
Elements in a cassette_properties.xml file	176
Step 2: Determine if additional configuration is required	178
Step 2A: (Optional) Use custom extension class if necessary	179
Step 3: Deploy your files to the target system	180
Typical installation examples	181
Migration considerations and steps	182
Using the migrate method	183
Running the paymentcassettemigrator script	183
Migrate interface in ICassetteConfigurator.java	184
Database considerations and steps	184
Uninstall considerations	185
Build a working cassette skeleton.	186
Build your administration objects.	187
Build your external view of administration objects	187
Implement your core protocol function.	190
Implement the basic CassetteOrder	190
Implement basic cassette payments	192
Implement CassetteBatch	193
Complete cassette payment.	195
Implement cassette credits	195
Complete the remaining transactional support	196
Create Cashier profiles (optional).	196
Understand platform-specific issues	197

Chapter 5. Testing your cassette **199**

Configuring your cassette	199
-------------------------------------	-----

Starting the WebSphere Commerce Payments user interface	200
Perform required configuration on the cassette	200
Creating a WebSphere Commerce Payments Merchant and authorizing a cassette.	200
Logging in as the Merchant Administrator	200
Creating an account	201
Managing payment processing	202
Using the Sale function to approve orders	203
Depositing payments	204

Settling batches.	204
Issuing a credit.	205
Viewing batch totals	207

Appendix. Notices 209

Trademarks	210
----------------------	-----

Index 213

Welcome!

This book explains how to write payment cassettes for the Payments component of IBM® WebSphere® Commerce, Version 5.5. It describes the WebSphere Commerce Payments cassette programming interface, cassette-specific programming or administrative considerations, and shows how the cassette implements the WebSphere Commerce Payments' various interfaces.

Note: WebSphere Commerce Payments was previously known as IBM WebSphere Payment Manager for Multiplatforms. Starting with version 3.1.3, the payments application was renamed to WebSphere Commerce Payments and references to the product were changed throughout this document. References to the former product may still appear in this document (such as in migration discussions) and apply to earlier releases of the product.

This book is for programmers who will develop payment cassettes for WebSphere Commerce Payments. Cassette developers should be experienced Java™ programmers with a strong background in the field of electronic payment processing.

Note: The Cassette Kit is provided "as-is" without warranties of any kind.

Payment cassettes process electronic payments using a specific payment protocol under control of WebSphere Commerce Payments. Since WebSphere Commerce Payments presents a single, well-defined payment processing application programming interface (API), adding new cassettes allows existing WebSphere Commerce Payments applications to process payments through new payment protocols with little or no integration effort.

Before reading this book or writing a payment cassette, the programmer should be very familiar with the material in the following documents:

- *WebSphere Commerce Installation Guide*
- *WebSphere Commerce Administration Guide*
- *WebSphere Commerce Payments Programming Guide and Reference*

Be sure to use the correct version of the documentation for the level of the WebSphere Commerce Payments component you are using.

If you are not familiar with the WebSphere Commerce Payments merchant administrative and programming interfaces, you should learn about them now.

This book is organized into these sections:

- Chapter 1, "Introducing WebSphere Commerce Payments", on page 1 provides a high-level description of the WebSphere Commerce Payments goals and architecture.
- Chapter 2, "Understanding the WebSphere Commerce Payments framework", on page 7 describes the WebSphere Commerce Payments framework in detail. The framework provides the environment in which each cassette executes. A description of the framework classes and interfaces used by cassettes is included.
- Chapter 3, "Designing your cassette", on page 157 describes the role, responsibilities and components of a cassette.


- Chapter 4, “Writing your cassette”, on page 171 describes how to write a cassette.
- Chapter 5, “Testing your cassette”, on page 199 describes ways you can test your cassette.

Occasionally, you will find references in this document to Javadoc. The Javadoc is located in the docs directory where you unzipped the Cassette Kit:
cassette_kit_unzip_dir:/docs/javadoc.


Conventions used in this book


This book uses the following highlighting conventions:

- **Boldface** type indicates commands or graphical user interface (GUI) controls such as names of fields, icons, or menu choices.
- Monospace type indicates examples of text you enter exactly as shown, file names, and directory paths and names.
- *Italic* type is used to emphasize words. Italics also indicate names for which you must substitute the appropriate values for your system. When you see the following names, substitute your system value as described.

 **Windows** indicates information specific to the Windows[®] operating environment.

 **AIX** indicates information specific to AIX[®].

 **Solaris** indicates information specific to the Solaris Operating Environment.

 **400** indicates information specific to the IBM iSeries[™] 400 (formerly called AS/400[®]).

 **Linux** indicates information specific to Linux.

References in this book to *workstation* platforms apply to Windows, AIX, Solaris, and Linux on Intel[®] platforms (not iSeries).

References to *Linux* apply to both Linux on Intel workstations and also to Linux on IBM eServer iSeries, pSeries[™], zSeries[™] and S/390[®] systems unless otherwise specified.

WC_installdir represents the following default installation paths for WebSphere Commerce:

 **AIX** /usr/lpp/WebSphere/CommerceServer*nn*

 **Linux**  **Solaris** /opt/WebSphere/CommerceServer*nn*

 **Windows** *drive:*\WebSphere\CommerceServer*nn*

 **400** /QIBM/ProdData/CommerceServer*nn*

Payments_installdir represents the following default installation paths for WebSphere Commerce Payments:

 /usr/lpp/WebSphere/CommerceServernn/payments

  /opt/WebSphere/CommerceServernn/payments

 drive:\WebSphere\CommerceServernn\payments

 /QIBM/ProdData/CommercePayments/Vnn

Additional information

More information about WebSphere Commerce and the Payments component is available from a variety of sources in different formats. The following are sources of WebSphere Commerce information:

- Online help
- Portable document format (PDF) files
- Web sites

Using the online help

The WebSphere Commerce online information provides information about customizing, administering, and reconfiguring WebSphere Commerce.

The WebSphere Commerce Payments online help provides information about how to use the graphical user interfaces associated with the Payments component. The Payments online help is available by clicking the question mark icon in the upper right corner of the user interface panel.

Locating the printable documentation

Some of the WebSphere Commerce online information is also available on your system in PDF files, which you can view and print using Adobe Acrobat Reader. In addition, WebSphere Commerce Payments documents are provided as PDF files. You can download the Acrobat Reader for free from the Adobe Web site at the following Web address:

<http://www.adobe.com>

PDF files can be accessed through the WebSphere Commerce online help and through the WebSphere Commerce Web site for product information.

Viewing the WebSphere Commerce Web site for product information

WebSphere Commerce product information is available at the WebSphere Commerce technical library Web site:

<http://www.ibm.com/software/commerce/wscom/library/lit-tech.html>.

Other WebSphere Commerce Payments documents and Web sites

The following documents provide information related to the Payments component of WebSphere Commerce:

- The *WebSphere Commerce Installation Guide* provides instructions on how to install and configure WebSphere Commerce Payments for your platform.

- The *WebSphere Commerce Administration Guide* contains conceptual information and shows how to configure WebSphere Commerce Payments using the Configuration Manager user interface.
- Cassette supplements are available for the various payment cassettes provided by IBM.

All documents are provided in Portable Document Format (PDF).

This book is a guide and reference to the WebSphere Commerce Payments framework and should be used in conjunction with the sample cassette LDBCARD, and its associated documentation. LDBCARD is a fully-functioning cassette, and is meant to serve as a skeleton upon which you can build your cassettes. The LDBCARD package includes a Cassette Developer Cookbook, which provides a step-by-step approach to developing a cassette. Download the LDBCARD package from the Web site from which you downloaded this book (see <http://www.ibm.com/software/webservers/commerce/payments/download.html> for the Cassette Developer's Toolkit).

Visit the following Web sites for more information about WebSphere Commerce Payments:

- <http://www.ibm.com/software/webservers/commerce/payment/> provides more information on the WebSphere Commerce payment-processing software, including information about the payment cassettes that are available for use with WebSphere Commerce Payments.
- <http://www.ibm.com/software/webservers/commerce/payments/support.html> provides current WebSphere Commerce Payments technical information and links to the latest WebSphere Commerce Payments documentation.
- <http://www.ibm.com/software/webservers/commerce/payment/paymentcards.html> provides information about WebSphere Commerce Payments cassette development.

WebSphere Commerce support and download information is available at the following Web sites:

- <http://www.ibm.com/software/commerce/wscom/support/index.html>
- <http://www.ibm.com/software/commerce/wscom/downloads/index.html>

What's new for release 5.5

All cassettes (IBM provided or third party) previously installed on WebSphere Commerce Payments, Version 2.2 or higher should continue to function after successfully installing WebSphere Commerce Payments, Version 5.5.

Before you install WebSphere Commerce Payments, refer to the *WebSphere Commerce Installation Guide* for your platform.

Directory file structure changes

Some changes were made to WebSphere Commerce Payments directory file structure. For example:

- The `eTillCal.zip` and `eTillClasses.zip` packages are now called `eTillCal.jar` and `eTillClasses.jar` respectively.
- The `Payments_installdir/include` subdirectory is now in `Payments_installdir/wc.mpf.ear/Payments.war/dtd`.
- With the exception of the `instances` subdirectory, the directory structure for iSeries now matches the structure for workstation platforms.

- The SampleCheckout application is in its own WAR file inside the WebSphere Commerce Payments EAR file. As a result, it is accessed through *hostname/webapp/SampleCheckout* rather than *hostname/webapp/PaymentManager/SampleCheckout*.

Installation and configuration changes

WebSphere Commerce Payments no longer has its own installation program. As a component of WebSphere Commerce, it is installed through the WebSphere Commerce installation program as described in the *WebSphere Commerce Installation Guide*. After installation, you must configure a Payments instance through the WebSphere Commerce Configuration Manager.

Using the Configuration Manager, you can configure and manage WebSphere Commerce instances, including instances of the Payments component. The Configuration Manager enables you to create, update, and delete Payments instances, start and stop them, change instance passwords, and add and remove cassettes for an instance. For more information about creating an instance, refer to the *WebSphere Commerce Installation Guide*. The *WebSphere Commerce Administration Guide* provides additional information about how to perform configuration tasks in WebSphere Commerce.

As a result of these changes, the way you install and configure cassettes you create has changed significantly. See “Installation and uninstallation considerations and steps” on page 173 for revised installation and configuration instructions for third-party cassettes.

Cassette migration

WebSphere Commerce Payments requires WebSphere Application Server Version 5, and as a result, you must configure and deploy your cassette differently than you did in the past. This document describes how to produce new cassettes for use with WebSphere Commerce Payments, and also describes how to migrate cassettes you have written in WebSphere Payment Manager Version 2.2.x or later format to WebSphere Commerce Payments Version 5.5 for use with WebSphere Application Server Version 5. For more information about migrating third-party cassettes, see “Migration considerations and steps” on page 182. Migration information for WebSphere Commerce is provided in the *WebSphere Commerce Migration Guide* for your platform.

If you need to migrate a cassette written to an earlier Payments format (such as Payment Server 1.2, or Payment Manager 2.1), you should contact IBM for migration assistance.

IBM-provided cassettes

The Cassette for SET™ and Cassette for CyberCash are no longer supported. The cassettes provided with WebSphere Commerce Payments consist of the following:

- OfflineCard Cassette
- CustomOffline Cassette
- Cassette for BankServACH
- Cassette for Paymentech
- Cassette for VisaNet

Default port removal

There is no longer a default port specified for WebSphere Commerce Payments (formerly, it was 80). Ports are specified through the WebSphere Commerce Configuration Manager.

Message and trace facility changes

WebSphere Commerce Payments now uses WebSphere Application Server message and trace facilities rather than its own facilities to generate system message and trace output. This change provides problem determination data in a more consistent fashion, making it easier for you to collect and understand the data in a WebSphere environment.

- Message changes include the following:

Messages can be viewed in the WebSphere Application Server administrative console and in the `activity.log` file in the `WAS_installdir/logs/instancename_CommercePayments` directory. Formerly, messages were written to the `PMError` file in the Payments logpath directory (`Payments_installdir/logs`) by default. Messages can be viewed with the WebSphere Log Analyzer.

Additionally, a WebSphere Commerce symptom database is available as a problem determination aid. Using the WebSphere Log Analyzer, you can view detailed information about Commerce system messages (including Payments messages) and view detailed explanations about the messages and suggested user response actions. More information about using the Log Analyzer with WebSphere Commerce logs, and the symptom database, is provided in the *WebSphere Commerce Administration Guide*. Also, refer to the WebSphere Application Server InfoCenter for complete details about the Log Analyzer.

- Trace changes include the following:

The Trace panel, which was formerly used to enable tracing, no longer appears in the WebSphere Commerce Payments graphical user interface.

To control which file the trace text is written to, use the WebSphere Application Server trace service to define where to output trace data, instead of the `PMTrace1.log` and `PMTrace2.log` files in the Payments logs directory. The `PMTrace` log files are no longer supported.

Chapter 1. Introducing WebSphere Commerce Payments

This section provides a technical overview of the WebSphere Commerce Payments framework for managing electronic payments. A complete solution is created when:

- WebSphere Commerce and its Payments component (called WebSphere Commerce Payments) is integrated with a business system requiring electronic payments.
- One or more payment cassettes are configured for use with WebSphere Commerce Payments.

Typically, we call the business system the *Merchant Server*, because the most common type of application that uses WebSphere Commerce Payments is shopping and catalog software for online merchants. However, it should be understood that the business system could be something as simple as an application that presents a shopper with a single "BUY" button to something as complex as an integrated order management, shipping and inventory system for a large company.

Each payment cassette implements a particular *payment protocol*, which is a predefined set of messages and processes that another online entity (often a financial institution or its agent) has defined to facilitate electronic payment processing. IBM currently offers these payment cassettes with WebSphere Commerce Payments:

- The CustomOffline Cassette supports processing of custom payment transactions such as COD, Bill-Me-Later or coupons that are often executed outside of WebSphere Commerce Payments.
- The OfflineCard Cassette supports online collection of credit card information for later use in manual execution and tracking of credit-card transactions.
- The Cassette for BankServACH supports processing of online electronic check payments using the BankServ payment gateway that interfaces with the Automated Clearing House Network (ACH).
- The Cassette for Paymentech supports online authorization and settlement of credit card and non-PIN based debit card payments.
- The Cassette for VisaNet supports processing of credit card transactions using the Vital Processing Services or First Horizon Merchant Services (FHMS) financial network.

This guide describes how third parties can create new payment cassettes that implement other electronic payment protocols.

WebSphere Commerce Payments features and goals

The WebSphere Commerce Payments framework provides the common function of a middleware server so that payment cassettes can focus strictly on implementing a particular payment protocol and the associated business logic. WebSphere Commerce Payments handles threading, caching, synchronization and access control. The framework also provides services to aid cassettes with communications, database access, event notification, scheduled work items, and error handling facilities.

The WebSphere Commerce Payments framework defines a *payment application programming interface (API)* that allows merchant shopping software to use a common mechanism for handling payments processed by diverse payment protocols. Differences between payment methods are masked by the API as much as possible. Where the differences cannot be masked, they are made easier to handle by the existence of common mechanisms.

The WebSphere Commerce Payments framework also defines an *object model* for payment-related data that provides the basic information and capabilities required for processing any type of electronic payments. The WebSphere Commerce Payments maintains persistent versions of each payment object in its SQL relational database. The framework works closely with each of its cassettes to ensure that the state of each framework object is correctly reflected in the database. This common payment data supports follow-on payment processing, such as refunds and settlement.

Payment cassettes built by third parties can be installed onto WebSphere Commerce Payments so that they are started when WebSphere Commerce Payments starts. Cassettes may also be dynamically stopped and started while WebSphere Commerce Payments continues operating.

The goal of the framework is to minimize the effort required to integrate new payment protocols with merchant software. Ideally, when a new cassette is added to a WebSphere Commerce Payments installation, merchant applications will be able to process payments through that cassette without any code changes for new parameters, error codes or other cassette-specific characteristics or behaviors. The framework provides a set of common functions, parameters and error codes for this purpose.

The cassette writer's job is to carefully map the functions and semantics defined by the Payment framework onto those provided by their payment protocol. Throughout this process, it is paramount to remember that the ultimate user of the cassette will be merchant software, and that the merchant software's view of the function is the view presented by the Payment framework.

In the end, the effort required to integrate a new payment cassette into an online merchant's system depends on how the payment cassette is built. If the cassette can implement its payment protocol using common command parameters and error codes, it is likely that the integration effort will be very small (if not eliminated altogether). If the cassette requires a lot of specialized parameters or institutes its own conventions and behaviors, the merchant's integration effort will be larger. Because of this, it is very important that cassette designers think clearly about the requirements that their cassette will place on merchant software and make every attempt to minimize those requirements.

For a complete description of the WebSphere Commerce Payments merchant programming interface, see the *WebSphere Commerce Payments Programming Guide and Reference*. For a description of the WebSphere Commerce administration policies and procedures, see the *WebSphere Commerce Administration Guide*.

WebSphere Commerce Payments architecture

WebSphere Commerce Payments consists of the following main components: the Payment Servlet, the User Interface Servlet, and a WebSphere Commerce Payments database as shown in Figure 1 on page 3.

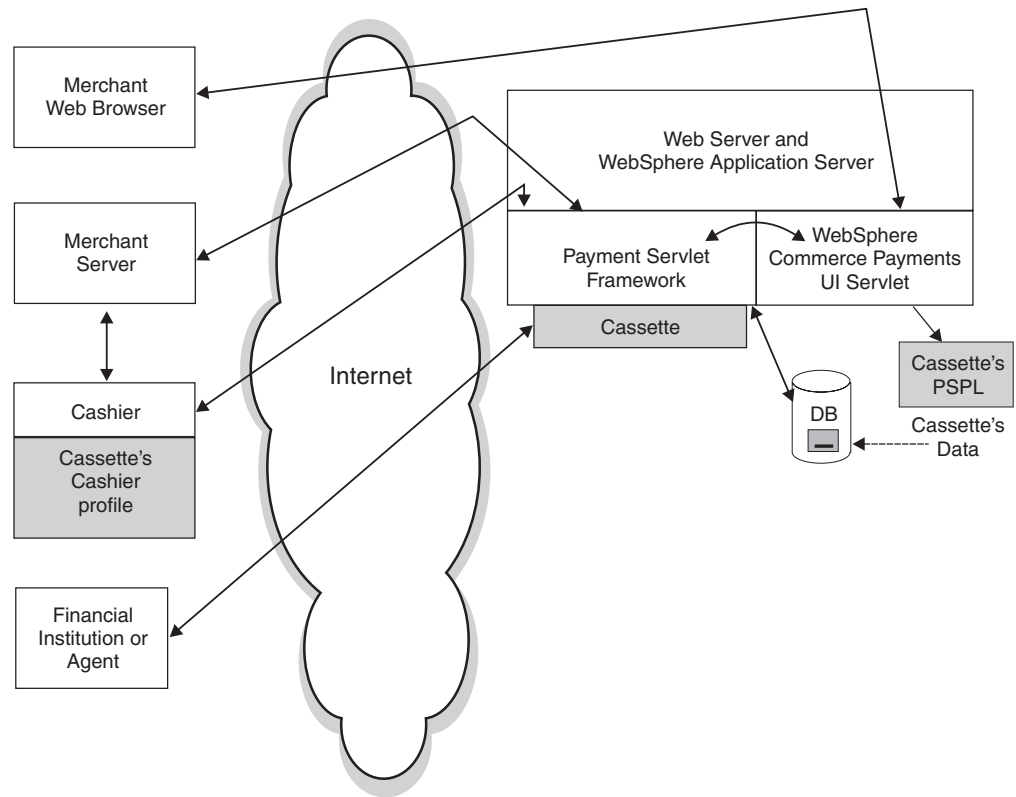


Figure 1. WebSphere Commerce Payments Architecture

Payment Servlet

The Payment Servlet performs the vast majority of the WebSphere Commerce Payments work. All commands are processed here, as are all protocol-specific messages between cassettes and the outside world. The Payment Servlet, when processing these requests, must also maintain the correct state of each object in the WebSphere Commerce Payments database.

This component also provides the merchant programming interface, which consists of a set of HTTP POST messages. The servlet runs under IBM's WebSphere Application Server's servlet environment and uses several of its services to communicate with client (merchant) software, authenticate each client, and manage user IDs and roles on which WebSphere Commerce Payments access controls are based.

Before sending commands to WebSphere Commerce Payments, an application or user must log in with a valid WebSphere Commerce Payments user ID. After logging in, the set of services available is controlled by the role assigned to the user ID through WebSphere's user management interface. These access controls ensure that each merchant's payment data is protected from other merchants and unauthorized users within the merchant site itself.

All of the WebSphere Commerce Payments API commands are built as standard HTTP POST messages. When a new command message arrives at the WebSphere Commerce Payments node, WebSphere receives and parses the HTTP message and then invokes the Payment Servlet to service the new request. All payment commands (AcceptPayment, Deposit, and so on),

administrative commands (for example, `ModifyMerchant` or `CreateAccount`) and query commands (for example `QueryOrders` or `QueryAccounts`) are processed within the servlet.

As each command is processed, an appropriate command response is built in an XML document using the IBM XML Java parser provided by WebSphere. Once the document is built, the servlet wraps it in an HTTP response message and then sends it back to the requesting user.

Cassette

A cassette is a software component consisting of Java classes and interfaces that support a particular form of payment protocol within the WebSphere Commerce Payments framework. Cassettes have both a query component and payment component. A cassette's query component is used to expose protocol-specific data to merchant applications. The protocol-specific data is combined with the framework data using standard constructs so that applications, including the WebSphere Commerce Payments user interface, can easily identify and interpret the data. Most of a cassette's work, however, is done by its payment processing component that runs within the Payment Servlet. For each type of request mentioned above, the associated cassette must perform at least a portion of the processing to complete the request.

User Interface Servlet

This servlet, which forms the WebSphere Commerce Payments user interface, generates the payment management and administration HTML pages that appear at a merchant's web browser. Likewise, this servlet processes the actions that are available through the HTML pages. The User Interface Servlet uses standard WebSphere Commerce Payments API commands to do its tasks, so to WebSphere Commerce Payments, it looks like any other application program.

Cassette's PSPL

Unlike the Payment Servlet, the User Interface Servlet never contains any running cassette code. If a cassette needs to expose its own data or administrative functions through the WebSphere Commerce Payments user interface, it describes the data and functions to the User Interface Servlet using a XML-based language, *Payment Server Presentation Language* (PSPL). PSPL tells the User Interface Servlet:

- How to build the user interface screens
- What data to use
- The characteristics of the data
- How to present each data item
- How to process input data (for administrative tasks).

DB All of the WebSphere Commerce Payments' configuration data and all of its financial data is recorded in the WebSphere Commerce Payments database. WebSphere Commerce Payments uses well-defined commit points to ensure the integrity of all the data related to each financial transaction and to allow for reliable checkpoint and restart operation of the server.

Cassette's Data

Because framework data only accounts for part of the complete representation of each transaction, cassettes also record their financial and configuration data in the WebSphere Commerce Payments database. The

cassette data is committed along with the framework data using the commit points described above. In fact, the cassettes actually choose and implement the commit points.

Cashier

The Cashier is WebSphere Commerce Payments code that can be invoked by client applications (such as merchant software) to simplify the process of creating WebSphere Commerce Payments orders and other WebSphere Commerce Payments commands. The Cashier uses XML documents called profiles that describe how commands such as orders should be created for a given cassette. This allows the client code writer to concentrate on integrating with WebSphere Commerce Payments in a generic way rather than having to write code that deals with cassette-specific information. WebSphere Commerce Payments orders can be created without using the Cashier (programs can still use the AcceptPayment and ReceivePayment APIs). The Cashier itself uses the Client API library (CAL) to send AcceptPayment, ReceivePayment, and Deposit commands to WebSphere Commerce Payments. The *WebSphere Commerce Payments Programming Guide and Reference* describes the Cashier and cashier profiles in more detail.

Cashier profile

Cashier profiles are XML documents that describe how WebSphere Commerce Payments commands should be created. Cashier profiles contain required WebSphere Commerce Payments and cassette parameters and specifications for how the Cashier supplies values for these parameters. Cashier profiles are stored in the profiles subdirectory of WebSphere Commerce Payments. Information about how to write a cashier profile is available in the *WebSphere Commerce Payments Programming Guide and Reference*.

Chapter 2. Understanding the WebSphere Commerce Payments framework

Before writing a WebSphere Commerce Payments cassette, you must understand the WebSphere Commerce Payments framework, whose goal is to provide:

- A single merchant programming interface, which allows merchant software to exploit new payment protocols with little or no change
- The internal structure into which cassettes must fit, function, and conform

To achieve this goal, the framework defines:

- “Framework object model”
- “Framework commands” on page 24

both of which can be augmented with cassette-specific information, as necessary.

The framework also provides services and features for cassettes and cassette developers that allow them to support the merchant programming interface and support the framework’s internal structure, as transparently as possible. These features are described in:

- “Responsibilities and services” on page 91
- “Framework Javadoc” on page 154

Framework object model

The WebSphere Commerce Payments framework defines the object model that is visible to merchant software and within which cassettes must define their own data. Because there is a very definite separation between the framework and cassettes, framework objects only define the general data that would apply to any electronic payment protocol. Cassettes may augment the framework objects with their own unique data using a set of framework mechanisms designed for this purpose.

The framework’s object model is divided into:

- The administrative object model, which defines the participants in financial transactions (such as merchants), operational entities (such as cassettes and the Payment Servlet itself) and relationships between the above types of objects. These objects are maintained in the database as well as in the Payment Servlet’s memory.
- The financial object model, which represents the transactions themselves (for example, orders, payments, and batches). These objects are maintained in the database as well as in the Payment Servlet memory.
- The merchant’s view of all of the above objects. This is a selective view of the framework objects that includes (potentially) selected cassette-specific data. These views are extracted from the database and assembled within the Payment Servlet in response to the WebSphere Commerce Payments set of Query commands.

Cassette writers must thoroughly understand this object model and must abide by the framework’s rules for accessing and extending its objects. These rules will be explained with the details of each portion of the object model.

The objects defined within this object model are created, updated, and deleted through the WebSphere Commerce Payments API commands. Having read the *WebSphere Commerce Payments Programming Guide and Reference*, you should already be familiar with the API commands. To understand how cassettes participate in the processing of each of these commands, see “Framework commands” on page 24.

Administrative object model

As previously described, the administrative object model defines the participants in financial transactions, operational entities and relationships between the above types of objects. These objects exist in the database and in the Payment Servlet’s memory. WebSphere Commerce Payments supports two different types of administrative objects:

Primary administrative objects

(or simply **primary objects**) are those that define a separate configurable entity. The framework defines several primary objects, such as PayServer, CassetteAdmin, MerchantAdmin, AccountAdmin, EventListener and so on. Cassettes may define their own primary objects, called MerchantCassetteObjects and SystemCassetteObjects, as needed. For example, many credit card-based cassettes choose to implement MerchantCassetteObjects to represent the concept of a credit card brand, because there is no analogous framework object.

The base class for all active instances of primary administrative objects is `com.ibm.etill.framework.admin.AdminObject`.

Cassette extension objects

are objects that provide the cassette-specific extensions to some of the framework’s primary administrative objects. For example, the framework’s AccountAdmin class contains the framework information describing an account, so the cassette extension to AccountAdmin contains the cassette’s own information regarding that account.

The term “cassette extensions” should not be confused with the Java term “extends”. Cassette extensions **do not imply** that a framework object should be extended in the Java sense of the word. Rather, a Cassette extension is a way for a cassette to define its own cassette-specific data to augment framework administrative classes.

There is no predefined base class within the Payment Servlet for cassette extensions. Nor are there any direct references within the framework’s primary administrative objects to their associated cassette extensions. Instead, administrative commands are forwarded to the cassette for processing. From there, the cassette decides how to handle the request, including which of its internal objects to update.

Primary administrative objects are created, modified and deleted through their respective CreateXxx, ModifyXxx and DeleteXxx (where Xxx represents Account, PaySystem, MerchantCassetteObject, or SystemCassetteObject) commands. The remainder of this section lists each of the framework’s primary administrative objects, beginning with the diagram below, which shows the relationships between these and other key objects. Cassettes may access any of the framework objects to obtain configuration information through the objects’ getter methods, but they may not modify any of them.

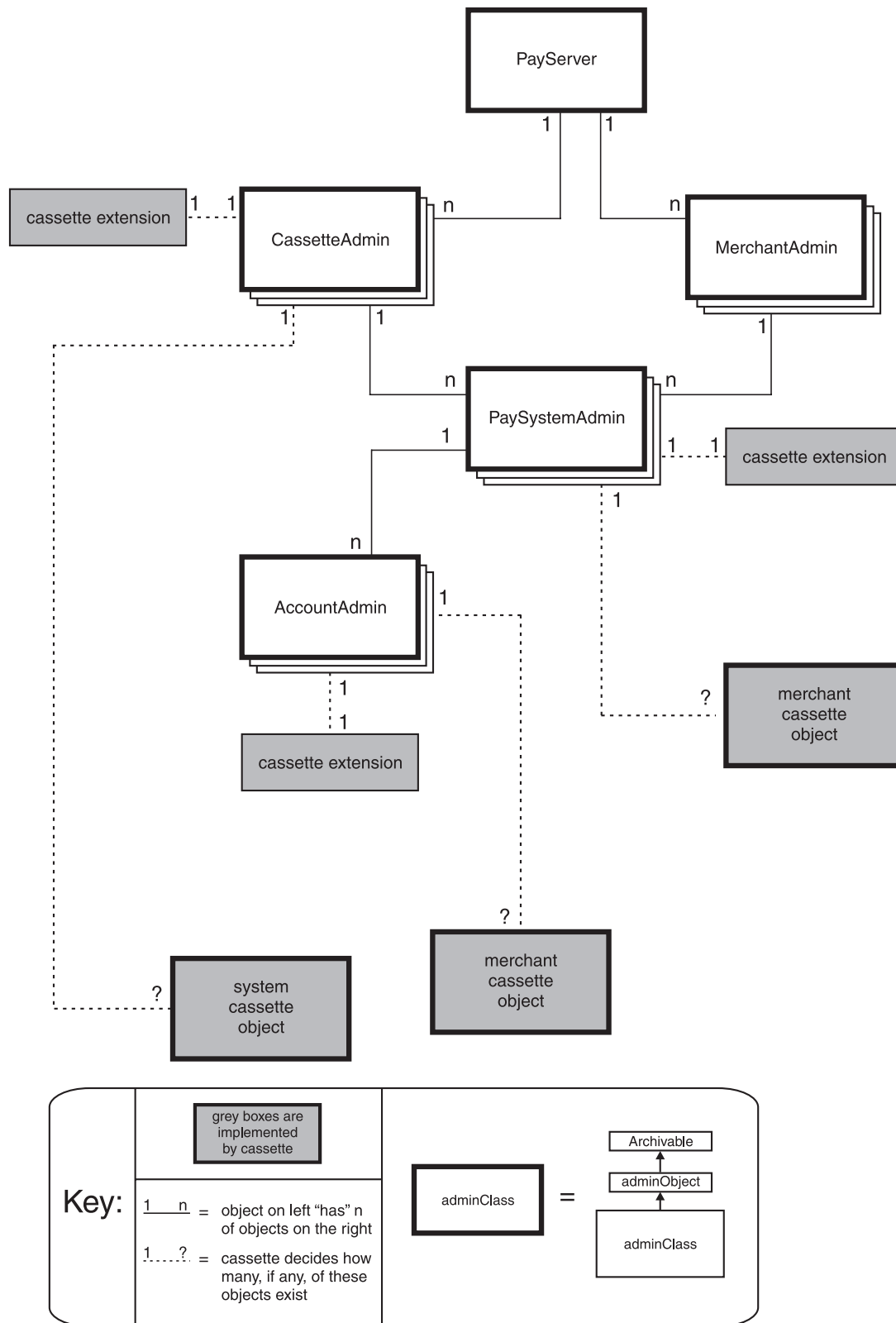


Figure 2. Framework administrative object model

PayServer

The PayServer object represents the Payment Servlet process. Only one of these objects exists in the servlet. This object contains server-wide

configuration parameters such as port values, and so on. For more information, see `com.ibm.etill.framework.admin.PayServer`.

CassetteAdmin

The `CassetteAdmin` object contains the framework's configuration information for a given cassette. There is one `CassetteAdmin` object for each active cassette in the WebSphere Commerce Payments instance. Cassettes may augment these objects with their own extensions. For more details, see `com.ibm.etill.framework.admin.CassetteAdmin`.

MerchantAdmin

The `MerchantAdmin` object contains the framework's configuration information for a given merchant. There is one `MerchantAdmin` object for each active merchant in the WebSphere Commerce Payments instance. For more details, see `com.ibm.etill.framework.admin.MerchantAdmin`.

AccountAdmin

Each `Account` object identifies a relationship between a merchant and a financial institution for a particular payment protocol. The framework maintains an `AccountAdmin` object that contains the relevant configuration information concerning the `Account`. Accounts are uniquely identified through the combination of the cassette name, the merchant number, and the account number. Cassettes may augment these objects with their own extensions. For more details, see `com.ibm.etill.framework.admin.AccountAdmin`.

PaySystemAdmin (Merchant Cassette Settings)

`PaySystemAdmin` is an association object. This object represents the association between a merchant and a cassette, which is called a "payment system" within WebSphere Commerce Payments (although the WebSphere Commerce Payments user interface calls this association the "Merchant Cassette Settings"). You can think of this object as the entity that allows a given merchant to use the services of a given cassette (hence, the association). Therefore, there is one `PaySystemAdmin` object for each merchant-cassette association. Cassettes may augment these objects with their own extensions. For more details, see `com.ibm.etill.framework.admin.PaySystemAdmin`.

Cassette Administrative Objects

The introductory material in this section already described the concept of cassette-specific primary administrative objects. These are instances of cassette-defined classes that extend `com.ibm.etill.framework.admin.AdminObject`. Each of these cassette classes is identified by a name that is used by merchant software and by the cassette to identify objects of such classes. Merchant software uses this name on the various `CreateXxx`, `ModifyXxx` and `DeleteXxx` commands that manipulate cassette administrative objects. The cassette uses the name to locate the appropriate object within its own internal data structures.

WebSphere Commerce Payments supports these types of cassette-specific primary objects:

SystemCassetteObject

is an administrative object that contain properties that apply across the cassette. `SystemCassetteObjects` are identified through a combination of the `ObjectName` and cassette name.

MerchantCassetteObject

is an administrative object that contains properties that apply only

to one merchant. MerchantCassetteObjects are identified through a combination of the merchant number, ObjectName and cassette name.

While cassettes may implement both types of administrative objects as they choose, there is no way for merchant software to directly query a SystemCassetteObject or a MerchantCassetteObject. Instead, such objects should be returned as ancillary data in support of queries for the framework objects to which they are logically related (this logical relationship is illustrated by the dotted lines in the diagram above).

For more details on how cassettes may expose their own administrative objects through the Exported Data Model, see “Exported Data Model: The user’s view of WebSphere Commerce Payments objects” on page 22 and “Query command processing” on page 90.

Financial objects and their states

As previously described, the financial object model defines the objects that represent financial transactions themselves. These objects exist in the database and in the Payment Servlet’s memory. Financial objects, upon which all payment-oriented commands operate, supported by WebSphere Commerce Payments include:

- Order
- Payment
- Credit
- Batch

The financial objects support follow-on payment processing such as refunds and settlement, as well as any other required financial operations. The financial objects allow merchant software to perform payment processing independent of any specific payment protocol.

The framework creates a persistent version of each object in the WebSphere Commerce Payments database that contains the initial data associated with the object as received from the merchant through a ReceivePayment or AcceptPayment command. From there, the state and contents of the objects change based on the commands that are performed on them. **Note: The cassette must maintain the framework state of these objects, since only the cassette can understand when the state has changed.** The state constants for these financial objects are defined in `com.ibm.etill.framework.payapi.PaymentAPIConstants`.

This diagram illustrates the relationship between the various framework financial objects. Notice that, unlike the framework’s administrative objects, each framework in-memory financial object contains a direct reference to the associated cassette object.

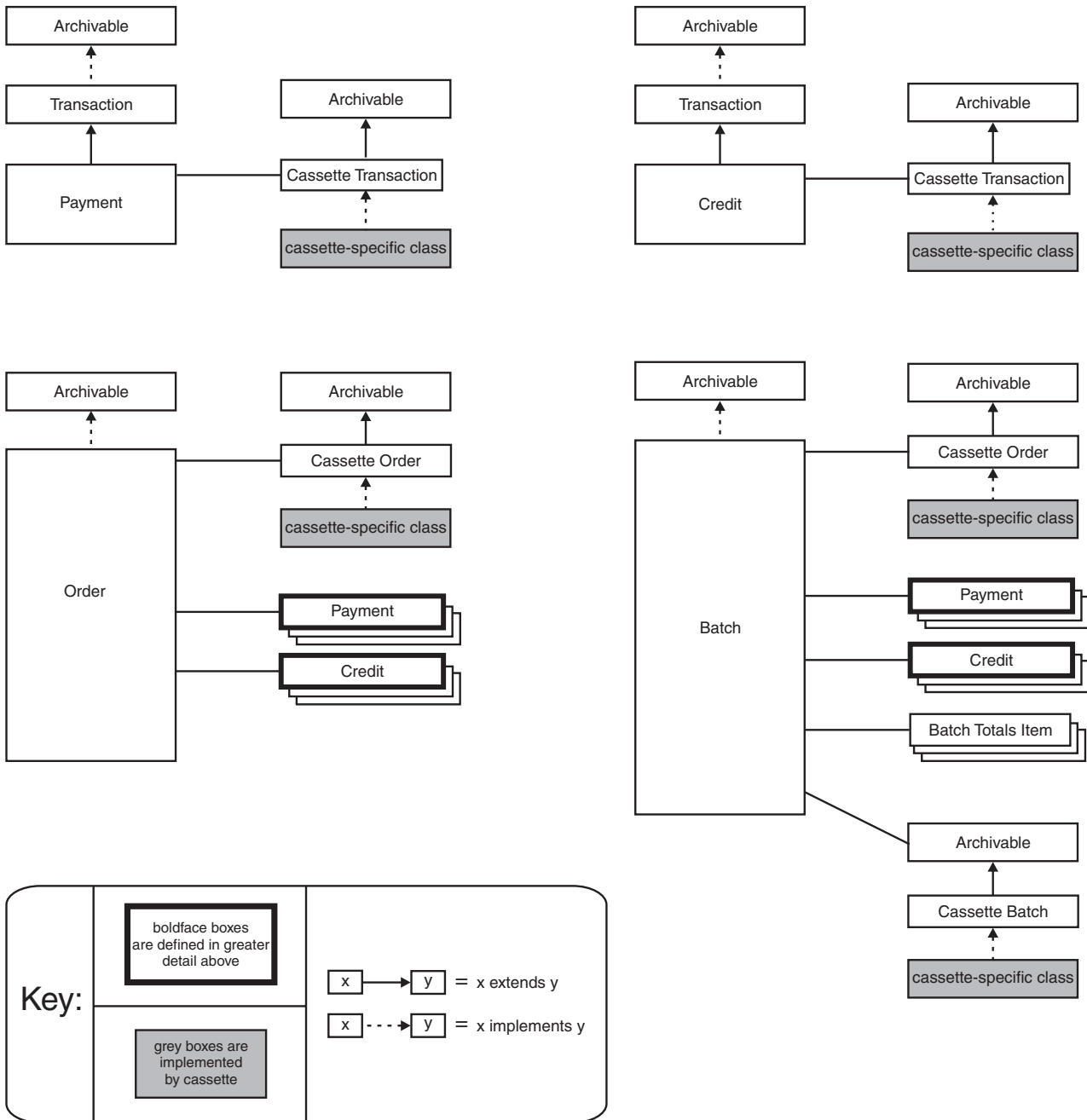


Figure 3. Relationships between framework financial objects

These sections describe these framework objects and the valid states associated with each framework object:

- “Orders” on page 13
- “Payments” on page 16
- “Credits” on page 18
- “Batches” on page 20

When a request arrives at WebSphere Commerce Payments, the framework locates or creates all of the appropriate in-memory objects. The cassette participates in object creation and retrieval by responding to requests from the framework. Once all payment objects are created or located, the framework calls the cassette to

process the request using these objects. For example, the cassette will be given the framework and cassette-specific order and payment data when asked to service a Deposit request.

Cassette writers are responsible for understanding these framework objects, and the framework data and states associated with each. Additionally, cassettes must be very careful to ensure that they do not inadvertently subvert the framework's ability to deliver on its guarantees concerning in-memory financial objects. Specifically, cassettes:

- **Should not** save references to these objects across requests
- **Should not** pass references to these objects to another thread, including service threads
- **Should not** keep its own cache or lists of in-memory objects
- **Should not** make its own "copies" of any of these in-memory objects

Long-lived references are dangerous because the cassette has no way of knowing when cached objects might be flushed from the cache or when administrative objects might be deleted or replaced. Creating copies of the objects would violate the framework's guarantee of a single in-memory representation of the data and the associated synchronization of operations on payment data.

Instead of keeping memory references or copies of these objects, cassettes should always ask the framework for the in-memory objects that they need to process each request. The framework provides all of the methods necessary to accomplish this. See "Order and batch caching" on page 91 for more information.

Orders

When discussing WebSphere Commerce Payments framework orders, it is important to understand the difference between the accepted definition of the word *order* (as it applies to commerce) and the WebSphere Commerce Payments definition of its `com.ibm.etill.framework.payapi.Order` object. The WebSphere Commerce Payments Order object *signifies agreement by the consumer to make a payment or payments using a single payment method*. The Order object contains payment information related to this agreement. It is important to understand there may not be a one-to-one correlation between a WebSphere Commerce Payments Order and what a merchant may consider to be an "order".

Given the above distinction, an Order represents all the instructions and information needed from the consumer (payer) in order for the merchant (payee) to collect money. The merchant may collect that money all at once or over a period of time, but the merchant never needs to go back to the consumer for additional information. All required information is contained in the Order.

Order objects are created when a `ReceivePayment` or `AcceptPayment` command is processed. Orders are uniquely identified by the combination of the merchant number and order number. Therefore, all order numbers for a given merchant number must be unique.

The Order Table (ETORDER) in the WebSphere Commerce Payments database contains the persistent instances of each Order object. There is one Order Table per WebSphere Commerce Payments instance and one row in the table per Order. While cassettes should always access Order objects through their Java representations (and not directly from the database), they **must** commit these objects to the database at the appropriate times, according to the needs of their payment protocol.

For a detailed description of its data members and methods, see `com.ibm.etill.framework.payapi.Order`.

Order states: All Orders exist in one of these states:

ORDER_RESET

The Order has been instantiated but has not yet been processed.

No commands are allowed for an order in this state since the `ReceivePayment` or `AcceptPayment` command for which the object was instantiated has not yet completed. Once command processing completes, the order will change to either `ORDER_REQUESTED` (for `ReceivePayment`) or `ORDER_ORDERED` (for `AcceptPayment`). (However, if, the payment cassette supports independent credit as described in “Independent credit” on page 16, the order will change to the `ORDER_REFUNDABLE` state for `AcceptPayment`). If automatic approval was requested on an `AcceptPayment` command, the order object will still pass through the `ORDER_ORDERED` state before the cassette can perform the automatic approval.

ORDER_REQUESTED

This state only occurs when the order was created in response to a `ReceivePayment` command. The payment protocol dictates whether or not this state is ever entered for each cassette.

This state indicates that the command was received from the merchant application and processed to the point of starting the purchase flows between consumer and WebSphere Commerce Payments, but those flows have not yet completed.

No commands are allowed for an order in this state. The cassette may place the order into either `ORDER_ORDERED` or `ORDER_REJECTED` state, once the purchase flows complete either successfully or unsuccessfully, respectively.

ORDER_ORDERED

Cassettes place an order in this state once all of the consumer’s financial information has been obtained and before any payments are closed.

Note: If the payment cassette is being designed to support independent credit, this order state is not used. See “Independent credit” on page 16 for more information.

For payment protocols that support this state, if the order was created through an `AcceptPayment` command, the Order should enter this state once the order has been successfully created. If the order was created through a `ReceivePayment` command, then it should enter this state when the consumer purchase flows (and any other processing required by the payment protocol) complete successfully.

When an order is in this state, these operations are allowed:

- Payments can be created or modified using `Approve` and `ApproveReversal` commands
- Existing approved payments can be captured using `Deposit` and `DepositReversal` commands
- The order can be canceled using the `CancelOrder` command, if all Payments and Credits are either in `VOID` or `DECLINED` state

ORDER_REFUNDABLE

An order is placed in this state by cassettes when either of the following occurs:

- After a payment has entered the `PAYMENT_CLOSED` state.
- The pre-existing state of the order is either `ORDER_RESET` or `ORDER_REQUESTED` and the cassette supports independent credit. This enables a merchant to issue credit for the order even if there is no payment.

The payment protocol dictates whether or not this state is ever entered for each cassette. If the protocol does not support credits, the cassette should not support this state. Otherwise, this state should be supported.

When an order is in this state, these operations are allowed:

- Assuming the payment protocol supports multiple payments per order, payments can be created or modified using `Approve` and `ApproveReversal` commands
- Assuming the payment protocol supports multiple payments per order, existing payments can be captured using `Deposit` and `DepositReversal` commands
- Credits can be created or modified using `Refund` and `RefundReversal` commands
- If all payment and credit objects belonging to the order are in their respective `CLOSED` states, then this order can be closed using the `CloseOrder` command.
- An order in `ORDER_REFUNDABLE` state can be cancelled through the `CANCELORDER` API command if there is no payment or credit for the order. (An order is considered to have a payment if the payment object associated with the order is in the `APPROVED`, `DEPOSITED`, `PENDING`, or `CLOSED` state.)

ORDER_REJECTED

This state only occurs when the order was created in response to a `ReceivePayment` command.

Cassettes should place an order in this state if an error occurs during the purchase flows between consumer and merchant for this order.

`CancelOrder` is the only command allowed for Orders in this state.

ORDER_PENDING

This state indicates that the cassette is waiting for an internal or external operation to complete for this Order.

The cassette must ensure that Order objects do not remain in pending state indefinitely, even after the account, payment system, cassette, merchant or payment server is stopped and restarted or after the system terminates unexpectedly. Ideally, the operation for which a pending object is waiting should be recovered or reinitiated when the account, payment system, cassette, merchant or payment server is restarted. At the very least, such pending objects should be placed into some other stable state during restart scenarios.

No commands are allowed for an order in this state.

ORDER_CANCELED

This state indicates that the merchant program or the cassette has voided this order before any payments or credits were completed. Orders are

canceled using the `CancelOrder` command, but cassettes may also put orders in this state if the payment protocol calls for it.

`CancelOrder` is the only command allowed for orders in this state.

ORDER_CLOSED

This state indicates that the merchant program has marked this order as no longer accepting payments or credits. Orders are closed using the `CloseOrder` command. For that command to succeed, the Order must have at least one closed payment or closed credit, and all payments and credits belonging to the order must be in their respective CLOSED states.

`CloseOrder` is the only command allowed for orders in this state.

Independent credit: Cassettes can be designed to support *independent credit*. The merchant server software can issue the `AcceptPayment` API command to create an Order, and then issue the `Refund` API command to issue an independent credit (that is, a credit that is not associated with a previous payment). For example, a customer desires a refund on an order, and the payment for that order has not been made. If the merchant decides to grant the customer a refund, the merchant can use the same order that the customer placed to issue a credit or create a new order.

When new orders are placed and the cassette supports independent credit, the order is placed in the `ORDER_REFUNDABLE` state rather than the `ORDER_ORDERED` state. Payment cassettes provided by WebSphere Commerce Payments that support independent credit include: the `OfflineCard` Cassette, `CustomOffline` Cassette, and `Cassette for VisaNet`.

Payments

A WebSphere Commerce Payments framework Payment object represents a "real world" payment made by a consumer to pay for all or part of an order. Payments are uniquely identified by the combination of the merchant number, order number, and payment number. Therefore, all payment numbers for a given order must be unique.

In many cases, all the money authorized for collection by the Order will be collected in a single payment. Some payment systems may allow the money authorized in one Order (that is, one set of Payment Instructions) to be collected in multiple portions. This "split payment" facility supports real world concepts like recurring payments and installment payments. The WebSphere Commerce Payments payment API supports the concept of split payment.

A Payment object must be created when either of these occur:

- An `Approve` command is received from a merchant application. The framework will create the payment prior to sending the approve request to the cassette.
- A `ReceivePayment` or `AcceptPayment` command is received with the `autoApprove` option requested (that is, with `APPROVEFLAG=1`).

If the `autoApprove` option is requested, there will never be a call to the `Approve` API. In this case, the cassette is expected to create a framework payment object during the `autoApprove` processing.

The framework allows zero or more Payments per Order, but the cassette ultimately determines the number of Payments allowed.

The Payment Table (ETPAYMENT) in the WebSphere Commerce Payments database contains the persistent instances of each Payment object. There is one Payment Table per WebSphere Commerce Payments instance and one row in the table per Payment. While cassettes should always access Payment objects through their Java representations (not directly from the database), they **must** commit these objects to the database at the appropriate times, according to the needs of their payment protocol.

For a detailed description of Payment's data members and methods, see `com.ibm.etill.framework.payapi.Payment`.

Payment states: All Payments exist in one of these states:

PAYMENT_RESET

This state indicates the payment has been instantiated, but has not yet been processed.

No commands are allowed for a payment in this state since the Approve command (or autoApprove processing) for which the object was instantiated has not yet completed.

Once the command processing completes, the cassette may place the payment into one of these states:

- `PAYMENT_APPROVED`, if the approve succeeds
- `PAYMENT_DECLINED`, if the financial institution or other approval authority rejects the request
- `PAYMENT_PENDING`, if the cassette has to wait for a response from the financial institution or retry the message before completing the request

PAYMENT_APPROVED

This state indicates some amount of funds have been approved, but those funds have not yet been deposited. When a payment is in this state, these operations are allowed:

- The approved amount may be modified using `ApproveReversal` commands. The payment protocol determines whether and how to support approval reversals. Choices are to not support such reversals at all, to only support full reversals (called a void), or to support partial reversals, in which the approval amount is changed to another non-zero value.
- A portion or all of the funds may be captured using the `Deposit` command (or through the automatic deposit processing option of the `Approve`, `AcceptPayment`, or `ReceivePayment` commands). Again, the decision to support complete or partial captures depends upon the payment protocol and cassette implementation.

PAYMENT_DEPOSITED

This state indicates that funds have been captured for this Payment, but the Batch to which the Payment belongs is not yet closed.

The payment protocol dictates whether or not this state is ever entered for each cassette. If the protocol does not enforce formal batch semantics for payments, the successful capture should move the payment directly into `PAYMENT_CLOSED` state. Otherwise, this state should be supported.

When a payment is in this state, these operations are allowed:

- The deposit may be voided using the `DepositReversal` commands. The payment protocol determines whether to support deposit reversals. Partial deposit reversals are not supported by the framework.

- The Payment may be closed through the successful closure of the batch to which the payment belongs. Depending upon the payment protocol and account or cassette configuration, batch closure may be explicitly requested by the merchant, implicitly initiated by the cassette, or be directed by the financial institution.

PAYMENT_CLOSED

This state indicates that the financial transaction is complete. Once a Payment is in this state, it cannot be modified.

PAYMENT_DECLINED

This state indicates that a financial failure occurred. Typically, cassettes only place payments into this state if an approval failed, but it may also be used in other cases if the payment protocol requires it.

The operations allowed on Payments in this state depend upon the payment protocol and the cassette implementation.

PAYMENT_VOID

Indicates that this Payment had been approved, but the full approval amount was subsequently reversed. Cassettes should support this state if the payment protocol allows for full approval reversals.

When a Payment is in this state, the only allowed operation is the Approve command, which is used to generate a new approval for the Payment.

PAYMENT_PENDING

This state indicates that the cassette is waiting for an internal or external operation to complete for this Payment. If a cassette places a Payment into this state, it should place the associated Order object into ORDER_PENDING state at the same time.

The cassette must ensure that Payment objects do not remain in PAYMENT_PENDING state indefinitely, even after the account, payment system, cassette, merchant or payment server is stopped and restarted, or after the system terminates unexpectedly. Ideally, the operation for which a pending object is waiting should be recovered or reinitiated when the account, payment system, cassette, merchant or payment server is restarted. At the very least, such pending objects should be placed into some other stable state during restart scenarios.

No commands are allowed for a Payment in this state.

PAYMENT_EXPIRED

This state indicates that the Order had been approved, but the payment subsequently expired. Cassettes should support this state if the payment protocol allows authorizations to expire.

When a payment is in this state, the only allowed operation is the APPROVEREVERSAL API command, which is used to generate a new approval for the payment or to perform a full or partial reversal and place the payment in the VOID state.

Credits

The Credit object represents one credit made against one of the existing Payment objects belonging to the associated Order object. Credits are uniquely identified by the combination of the merchant number, order number, and credit number. Therefore, all credit numbers for a given order must be unique.

The framework creates a Credit object when a Refund command is received from a merchant application. The framework allows zero or more Credits per Order, but

the cassette ultimately determines the number of Credits allowed. The general rule is that the total amount of all Credits for a given Order must be less than or equal to the total amount of the Payments associated with that Order. Again, it is up to the cassette to enforce any rules imposed by the payment protocol regarding the total amount of credits allowed against any given order.

The Credit Table (ETCREDIT) in the WebSphere Commerce Payments database contains the persistent instances of each Credit object. There is one Credit Table per WebSphere Commerce Payments instance and one row in the table per Credit. While cassettes should always access Credit objects through their Java representations (not directly from the database), they **must** commit these objects to the database at the appropriate times, according to the needs of their payment protocol.

For a detailed description of Credit's data members and methods, see `com.ibm.etill.framework.payapi.Credit`.

Credit states: All Credits exist in one of these states:

CREDIT_RESET

This state indicates credit has been instantiated, but has not yet been processed.

No commands are allowed for an order in this state since the Refund command for which the object was instantiated has not yet completed.

Once the command processing completes, the cassette may place the Credit into one of these states:

- **CREDIT_REFUNDED**, if the refund succeeds and the Batch to which the Credit belongs is still open
- **CREDIT_CLOSED**, if the refund succeeds and the Batch to which the Credit belongs closed when this refund completed
- **CREDIT_DECLINED**, if the financial institution or other authority rejects the request
- **CREDIT_PENDING**, if the cassette has to wait for a response from the financial institution or retry the message before completing the request.

CREDIT_REFUNDED

This state indicates that the refund has been processed by the financial institution, but the Batch to which the Credit belongs is not yet closed.

Note that the payment protocol dictates whether or not this state is ever entered for each cassette. If the protocol does not enforce formal batch semantics for Credits, the successful refund should move the Credit directly into **CREDIT_CLOSED** state. Otherwise, this state should be supported.

When a Credit is in this state, these operations are allowed:

- The refunded amount may be voided using `RefundReversal` commands. The payment protocol determines whether to support refund reversals. Partial refund reversals are not supported by the framework.
- The Credit may be closed through the successful closure of the Batch to which the Credit belongs. Depending upon the payment protocol and account or cassette configuration, batch closure may be explicitly requested by the merchant, implicitly invoked by the cassette, or be directed by the financial institution.

CREDIT_CLOSED

This state indicates that the financial transaction is complete. Once a Credit is in this state, it cannot be modified.

CREDIT_DECLINED

This state indicates that a financial failure occurred.

The operations allowed on Credits in this state depend upon the payment protocol and the cassette implementation.

CREDIT_VOID

This state indicates that the refund had been issued, but the full refund amount was subsequently reversed. Cassettes should support this state if the payment protocol allows refund reversals.

When a Credit is in this state, the only allowed operation is the Refund command, which is used to generate a new refund transaction under the same credit number.

CREDIT_PENDING

This state indicates that the cassette is waiting for an internal or external operation to complete for this Credit. If a cassette places a Credit into this state, it should place the associated Order object into ORDER_PENDING state at the same time.

The cassette must ensure that Credit objects do not remain in CREDIT_PENDING state indefinitely, even after the account, payment system, cassette, merchant or payment server is stopped and restarted, or after the system terminates unexpectedly. Ideally, the operation for which a pending object is waiting should be recovered or reinitiated when the account, payment system, cassette, merchant or payment server is restarted. At the very least, such pending objects should be placed into some other stable state during restart scenarios.

No commands are allowed for a Credit in this state.

Batches

A Batch is a collection of Payments and Credits to be processed as a group by a financial institution (much like multiple database operations can be grouped together within a single work unit for an eventual commit or rollback operation). When a Batch is closed, all of the transactions within the batch are completed and made permanent.

Not all payment protocols support or require batch semantics. However, even in these cases you are strongly encouraged to use the batch constructs provided by WebSphere Commerce Payments. Doing so ensures a consistent programming and user interface across all payment protocols for merchant software and end users. Even when the payment protocol does not support batches, you can still build your own "local" batches inside of WebSphere Commerce Payments to logically group your transactions for some period of time that is either determined by the user or specified somewhere into the cassette's configuration. This will allow users to perform their daily maintenance operations (such as closing the day's batches) and view their transaction data using the same WebSphere Commerce Payments user interface totals and summary views as they do for all other payment protocols.

Batches are uniquely identified by the combination of the merchant number and batch number. Therefore, all batch numbers for a given merchant must be unique.

Each Batch is further associated with a given account. The framework allows an account to have zero or more Batches, but the actual number allowed is ultimately up to the cassette.

The creation of Batch objects is ultimately the responsibility of cassettes since batch policies vary for each payment protocol. With respect to the API, there are two primary approaches to batch creation: explicit batch management or implicit batch management.

Explicit style: Explicit style can only be used when the merchant controls the batch (that is, the merchant is responsible for opening and closing of batches). All of the merchant decisions (opening and closing batches, assigning payments and credits to batches) are exported through the API to the merchant software. The merchant opens batches using the `BatchOpen` command, closes batches using the `BatchClose` command, and must specify a batch number on all deposit and refund commands.

If at all possible, avoid using this style because it does not work well with the WebSphere Commerce Payments user interface and because it burdens the merchant software with details that can most likely be avoided.

Implicit style: Implicit style can be used whether the merchant controls the batch or the financial institution controls the batch. The merchant does not issue `BatchOpen` commands and does not specify a batch number on `Deposit` or `Refund` commands. If the financial institution controls the batch, WebSphere Commerce Payments forwards requests to the financial institution, which in turn manages the batches. If the merchant controls the batch, WebSphere Commerce Payments manages the batches, as necessary. When implicit style is used:

1. WebSphere Commerce Payments framework receives a `Deposit` command and forwards it to the cassette using `service()`.
2. Cassette determines if there is currently a batch open for the specified account.
3. If there is, then deposit the payment in the existing open batch.
4. If there is not, then ask the framework to create a batch (`Supervisor.createBatch()`) and deposit the payment in the newly created batch. The framework will generate a batch number for the batch. This framework creation of the batch is considered an implicit `BatchOpen`.

You are strongly encouraged to use the Implicit style. As noted above, implicit batch management can be used regardless of which entity (merchant or financial institution) the payment protocol indicates is in control of the batch. This style works well with the WebSphere Commerce Payments user interface and hides details from merchant software that it would otherwise never need to consider.

The Batch Table (ETBATCH) in the WebSphere Commerce Payments database contains the persistent instances of each Batch object. There is one Batch Table per WebSphere Commerce Payments instance and one row in the table per Batch. While cassettes should always access Batch objects through their Java representations (not directly from the database), they **must** commit these objects to the database at the appropriate times, according to the needs of their payment protocol.

For a detailed description of Batch's data members and methods, see `com.ibm.etill.framework.payapi.Batch`.

Batch states: All Batches exist in one of these states:

BATCH_OPENING

This state indicates that the Batch object has been instantiated, but has not yet completed open processing. This occurs in response to a BatchOpen command (if this is a merchant-controlled batch) or when a batch is opened implicitly by the cassette.

No operations are allowed against a Batch in this state.

Once the open processing completes successfully, the cassette will place this Batch into BATCH_OPEN state.

BATCH_OPEN

Batches exist in this state between the time that the batch opens (either explicitly or implicitly) successfully and when the batch close processing begins.

When a Batch is in this state, these operations are allowed:

- Payments may be added in response to Deposit commands,
- Payments may be removed in response to DepositReversal commands,
- Credits may be added in response to Refund commands,
- Credits may be removed in response to RefundReversal commands,
- the Batch may be closed, either in response to a BatchClose command or as a result of conditions in the cassette or at the financial institution.

BATCH_CLOSING

This state indicates that the batch close process has begun, but is not yet complete. This occurs in response to a BatchClose command or when conditions inside the cassette or at the financial institution require that the Batch be closed.

The cassette must ensure that Batch objects do not remain in closing state indefinitely, even after the account, payment system, cassette, merchant or payment server is stopped and restarted or after the system terminates unexpectedly. Ideally, the operation for which a pending object is waiting should be recovered or reinitiated when the account, payment system, cassette, merchant or payment server is restarted. At the very least, such pending objects should be placed into some other stable state during restart scenarios.

No operations are allowed on a Batch in this state. The Batch will remain in this state until batch settlement completes.

BATCH_CLOSED

This state indicates that the batch closing procedure has completed successfully. In general, this will be when the settlement phase completes.

DeleteBatch is the only command allowed for Batches in this state.

Exported Data Model: The user's view of WebSphere Commerce Payments objects

Now that we've looked at the basic types of objects that WebSphere Commerce Payments maintains, we'll examine the mechanism through which these objects are exposed to applications and end users.

Applications need to view the WebSphere Commerce Payments objects in order to manage not only their own transactions, but also WebSphere Commerce Payments itself. However, in exposing this data to applications, several important points must be considered:

- While the WebSphere Commerce Payments framework and its cassettes maintain all of their data in the WebSphere Commerce Payments database, not all of that data is suitable for exposure to applications. Therefore, the framework and cassettes must be able to selectively expose only the data that is essential to applications' ability to accomplish their tasks.
- When dealing with financial data or configuration data that applies to only a single merchant, only the data belonging to that merchant should be exposed. Stated another way, access controls must be enforced on requests for merchant-specific data.
- The schema of the WebSphere Commerce Payments database must be able to change without impacting the application's view of that data. This is essential to ensure compatibility across service updates and new releases of the framework and cassettes. **Therefore, applications should never be required or allowed to access the WebSphere Commerce Payments database directly.**

To facilitate this selective, secure, and database-independent view of WebSphere Commerce Payments data, WebSphere Commerce Payments provides an Exported Data Model (XDM). The XDM is accessed through the WebSphere Commerce Payments's Query commands that return XML documents that contain representations of the requested object or objects. These Query commands are processed completely within the Payment Servlet. For a complete description of the XML representation of WebSphere Commerce Payments objects, see the *WebSphere Commerce Payments Programming Guide and Reference*.

As previously mentioned, cassettes may also export portions of their data through XDM. To do this, the cassette writer must first decide which of its data items needs to be exposed to merchant applications in order to adequately perform payment processing and then methods need to be written to expose this data and convert it into predefined XML elements. The cassette must then provide a set of classes that will run under the Payment Servlet under control of the framework in that process. For each object the cassette will expose through the query commands, it must implement:

- A class that extends `com.ibm.etill.framework.xdm.QueryRequest`. This class contains the methods required to efficiently extract all of the associated cassette extension objects associated with a given set of related framework objects. This class also contains methods that combine the cassette-specific query results with those of their corresponding framework objects.
A `QueryRequest` method is also available to indicate whether the query should return sensitive data to the user based on the user's minimum role, or hide it by masking the data with asterisks (see "Protecting sensitive data" on page 29 for more information).
- A class that represents the (selective) external view of the cassette object. The constructor for this class should take a `JDBC ResultSet` as input as each instance of this class will be the Java representation of one retrieved cassette object. "View" classes that represent `MerchantCassetteObjects` and `SystemCassetteObjects` extend `PSServerAdminObject`. For objects that simply extend framework objects, there is no common base class.

For a description of the internal control flow that takes place for query commands, see "Query command processing" on page 26.

Another important point to remember is that the objects in the exported object model contain various attributes that are intended specifically for use by the

WebSphere Commerce Payments user interface. Many of these attributes can be set or modified by the cassette to affect the eventual presentation of its data through the user interface.

Framework commands

The WebSphere Commerce Payments framework defines a payment application programming interface (API) that allows merchant software to use a common mechanism for handling all payment methods. Differences between payment methods are masked by the API as much as possible. Where the differences cannot be masked, they are made easier to handle by the existence of common mechanisms.

The WebSphere Commerce Payments API is documented from the merchant software's point of view in the *WebSphere Commerce Payments Programming Guide and Reference*. Study the *Programming Guide and Reference* to thoroughly understand the commands and the corresponding data provided with each command. These commands and data will be forwarded to your cassette after the framework has performed the associated framework actions.

After completing your study of the API commands, you must create a meaningful mapping between the WebSphere Commerce Payments command set and the functions available in the target payment protocol. During this process, and during the entire cassette development process, remember that the user of the cassette is merchant software, and the merchant software's view of the protocol is through the framework's WebSphere Commerce Payments payment API. Therefore, it is important to create this mapping from the viewpoint of the WebSphere Commerce Payments payment API, rather than trying to expose each and every feature of the payment protocol through the framework. For example, ask the question "what should happen if a `DepositReversal` is made to my cassette?" instead of "how can I expose my protocol's `voidAndRepurchase` function through the WebSphere Commerce Payments API?"

The key point is that since the merchant software will be coded to the single WebSphere Commerce Payments API, the cassette must attempt to work within the framework rather than attempting to change the look and feel of the framework to expose every possible feature of the protocol. Stated another way, just because a function is available in the protocol does not necessarily mean that it *has* to be used. Conversely, in many cases, the functions provided by the WebSphere Commerce Payments API will not have an analog in the payment protocol (for example, not all protocols will support the concept of a deposit reversal). For these cases, the payment cassette should not support the WebSphere Commerce Payments API call and should return an error code that indicates that the function is not supported. See "Protocol data: Cassette-specific command parameters" on page 31 for more information about return code constants that should be returned in the case of errors.

Attention must also be paid to the command parameters upon which the cassette will operate. Cassette writers should make every attempt to operate completely on the data provided by the framework parameters defined for each framework command. Again, given the wide variety of existing and potential payment protocols, it is clear that some protocol-specific parameters will be required at various times. Where a cassette does require its own unique parameters, the framework provides a simple, yet powerful, mechanism called *protocol data*.

IBM actively promotes the coordination of cassette-specific parameters and error codes across payment cassettes to provide the most consistent merchant programming interface possible, while still allowing cassettes to obtain protocol-specific data as needed. This is done by working with cassette developers to maintain a well-known (and frequently updated) list of commonly-used protocol data parameters and error codes.

Command processing overview

WebSphere Commerce Payments provides three "classes" of commands:

- Administrative commands (the "admin API")
- Payment processing commands (the "payment API"), and
- Query commands (the "query API")

Query commands provide read-only access to WebSphere Commerce Payments data. Payment and administrative commands can change the values of WebSphere Commerce Payments data. All WebSphere Commerce Payments results are returned in XML-based PSApiResult documents. If you are not using the Java CAL API, you will need an XML parser (a validating parser is preferred) and access to the `IBMPaymentServer.dtd` Document Type Definition (DTD) to process these documents. The DTD can be obtained from a WebSphere Commerce Payments installation. The XML parser can be obtained from the WebSphere Application Server installation.

Payment and Administrative command processing

The main interface class between the framework and a cassette in the Payment Servlet is the following abstract class:

```
com.ibm.etill.framework.cassette.Cassette
```

In Java terms, each cassette extends this class to provide the basic services that the framework expects from a cassette. When the framework loads the cassette, one instance of the cassette's `Cassette` subclass is created and this becomes the object to which all inbound requests to that cassette are routed. Specifically, the framework calls the cassette object's `service` method to process inbound requests.

To process a payment or administrative command, these events occur:

1. An application program builds a command in a URL encoded HTTP POST message and sends it to the WebServer on the WebSphere Commerce Payments host.
2. The WebSphere Application Server (Application Server) receives the HTTP POST message and parses the keyword-value pairs into `com.ibm.etill.payapi.ParameterTable` (a Java hash table), performing any necessary URL decoding along the way. The Application Server then invokes the Payment Servlet with the new request.
3. The Payment Servlet determines if the user is authorized for the specified command.
4. The Payment Servlet routes control to the framework's command processor.
5. For payment commands, the framework command processor builds an appropriate API Request (or derivative) object that represents the command. For administrative commands, the framework command processor builds an appropriate AdminRequest (or derivative) object that represents the command.
6. The framework performs its own processing for the command. This typically involves operations only on framework-owned objects.

7. Once the framework-level processing is complete, the framework locks a specific set of objects in order to ensure thread safety and data integrity while the cassette performs its responsibilities for the request. The choice of objects to lock and the type of lock obtained for each object is determined by the command type.
8. Once all of the object locks are obtained, the framework calls the appropriate Cassette object's **service** method to perform the necessary cassette-specific processing for the request. Depending upon which command is being processed, the cassette will operate on any number of cassette-owned and possibly framework-owned objects, and may assist in directing any other operations required to complete the processing of the request.
9. When the cassette completes its processing, it returns to the framework with success or failure codes as appropriate.
10. The framework returns the response to the Payment Servlet, which puts the response information into an XML document, encapsulates the XML document response within an HTTP response message, and sends the message back to the merchant application.

Query command processing

The main interface class between the framework and a cassette in the Payment Servlet is the following class:

com.ibm.etill.framework.cassette.query.CassetteQuery

In Java terms, each cassette extends this class to provide the basic services that the framework expects from a cassette. When the framework loads the cassette, one instance of the cassette's `CassetteQuery` subclass is created and this becomes the object to which all inbound query requests to that cassette are routed. Specifically, the framework calls the cassette object's **query** method to process inbound query requests.

To process a Query command, these events occur:

1. A merchant application builds a command message and sends it to the WebSphere Commerce Payments host's web server in a URL encoded HTTP POST message.
2. The WebSphere Application Server (Application Server) receives the HTTP POST message and parses the keyword-value pairs into `com.ibm.etill.payapi.ParameterTable` (a Java hash table), performing any necessary URL decoding along the way. Application Server then invokes the Payment Servlet with the new request.
3. The Payment Servlet determines if the user is authorized for the specified command.
4. The Payment Servlet routes control to the framework's command processor.
5. The Payment Servlet creates an appropriate `QueryRequest` (or derivative) framework object that represents the command. The `QueryRequest` contains the criteria for the query. It also specifies whether data, considered sensitive by the cassette, should be returned to the user or hidden (masked) through the use of asterisks. (See "Protecting sensitive data" on page 29 for a description of this processing.) The `QueryRequest` will build and execute the appropriate SQL query against the WebSphere Commerce Payments database (framework tables) to fulfill the query request.
6. Once the framework-level processing is complete, the Payment Servlet then asks the `QueryRequest` to issue the query to the appropriate cassette(s) through the cassette query, so that the cassette(s) can add cassette-specific fields to the

result. Cassettes should build and execute the appropriate SQL query against the WebSphere Commerce Payments database (Cassette tables) to fulfill the request.

7. When the cassette completes its processing, the Payment Servlet creates an appropriate QueryResponse (or derivative) framework object that represents the results from the query command.
8. An XML document is then created based on the results that are in the QueryResponse. The XML document is returned to the Payment Servlet, which then encapsulates the XML document within an HTTP response message and sends the message back to the merchant application.

Request classes and their hierarchy

As described in the previous sections, the framework calls a central method in the cassette (the Cassette object's service method and the CassetteQuery object's query method) to process each command. The form in which the commands are passed to the cassette is that of a *request object*.

All Payment and Administrative request objects extend from the `com.ibm.etill.framework.cassette.CassetteRequest` base class.

The framework calls the Cassette object's service method to process each API Request. The service method can be considered the cassette's main "entry point" for processing user- or cassette-initiated requests. The framework calls the Cassette object's service method to process these types of requests:

- API requests, which represent an API command that was issued by merchant or administrative software. Each payment-oriented command is represented through its own request class. Administrative requests are represented by a single common request class. All commands extend the `com.ibm.etill.framework.cassette.APIRequest` subclass of `CassetteRequest`.
- Protocol requests, which are messages defined by the cassette to support its payment protocol. Since these are defined by the cassette, the framework supports a single class, `ProtocolRequest`, to represent them, which cassettes subclass as necessary.
- `CassetteWorkItem` requests, which are work items scheduled by the cassette (as necessary). As with protocol requests, the framework supports a single `CassetteWorkItem` class, which cassettes subclass as necessary.

While the majority of our discussion here will be centered around commands (that is, `APIRequest` objects), we will also discuss `ProtocolRequests` and `CassetteWorkItems` so that you are familiar enough with each to understand how to apply them to your cassette. The request class hierarchy is as follows:

```
java.lang.Object
|
+--com.ibm.etill.framework.cassette.CassetteRequest
|
+--com.ibm.etill.framework.cassette.APIRequest
|
|   +--com.ibm.etill.framework.cassette.AdminRequest
|   +--com.ibm.etill.framework.cassette.BatchRequest
|       |
|       +--com.ibm.etill.framework.cassette.BatchCloseRequest
|       +--com.ibm.etill.framework.cassette.BatchOpenRequest
|       +--com.ibm.etill.framework.cassette.BatchPurgeRequest
|       +--com.ibm.etill.framework.cassette.DeleteBatchRequest
|
+--com.ibm.etill.framework.cassette.OrderRequest
```

```

|--com.ibm.etill.framework.cassette.AcceptPaymentRequest
|--com.ibm.etill.framework.cassette.CancelOrderRequest
|--com.ibm.etill.framework.cassette.CloseOrderRequest
|--com.ibm.etill.framework.cassette.PaymentTransactionRequest
|
|--com.ibm.etill.framework.cassette.ApproveRequest
|--com.ibm.etill.framework.cassette.ApproveReversalRequest
|--com.ibm.etill.framework.cassette.DepositRequest
|--com.ibm.etill.framework.cassette.DepositReversalRequest
|--com.ibm.etill.framework.cassette.RefundRequest
|--com.ibm.etill.framework.cassette.RefundReversalRequest
|--com.ibm.etill.framework.cassette.ReceivePaymentRequest
|--com.ibm.etill.framework.cassette.CassetteWorkItem
|--com.ibm.etill.framework.cassette.ProtocolRequest

```

Cassettes you write must use command tokens to validate and process administrative requests. The tokens identify the type of administrative request that is being made. Table 1 lists the command tokens used in the API request objects passed to the Cassette object's service method.

Table 1. Command tokens used in WebSphere Commerce Payments API requests. When debugging your cassette, you can find the meaning of a command token value from this list, or refer to `com.ibm.etill.framework.payapi.PaymentAPIConstants`.

Command token	Value
ABOUT_TOKEN	1280
ACCEPTPAYMENT_TOKEN	1209
APPROVE_TOKEN	1201
APPROVEREVERSAL_TOKEN	1204
BATCHCLOSE_TOKEN	1208
BATCHOPEN_TOKEN	1207
BATCHPURGE_TOKEN	1224
CANCELORDER_TOKEN	1222
CASSETTECONTROL_TOKEN	1219
CLOSEORDER_TOKEN	1221
CREATE_ACCOUNT_TOKEN	1259
CREATE_MERCHANT_TOKEN	1253
CREATE_MERCHANTCASSETTEOBJECT_TOKEN	1265
CREATE_MEREVENTLISTENER_TOKEN	1268
CREATE_PAYSYSTEM_TOKEN	1256
CREATE_SNMEVENTLISTENER_TOKEN	1306
CREATE_SYSTEMCASSETTEOBJECT_TOKEN	1262
DELETE_ACCOUNT_TOKEN	1261
DELETE_MERCHANT_TOKEN	1255
DELETE_MERCHANTCASSETTEOBJECT_TOKEN	1267
DELETE_MEREVENTLISTENER_TOKEN	1270
DELETE_PAYSYSTEM_TOKEN	1258
DELETE_SNMEVENTLISTENER_TOKEN	1308
DELETE_SYSTEMCASSETTEOBJECT_TOKEN	1264

Table 1. Command tokens used in WebSphere Commerce Payments API requests (continued). When debugging your cassette, you can find the meaning of a command token value from this list, or refer to `com.ibm.etill.framework.payapi.PaymentAPIConstants`.

Command token	Value
DELETEBATCH_TOKEN	1223
DEPOSIT_TOKEN	1202
DEPOSITREVERSAL_TOKEN	1205
MODIFY_ACCOUNT_TOKEN	1260
MODIFY_CASSETTE_TOKEN	1252
MODIFY_MERCHANT_TOKEN	1254
MODIFY_MERCHANTCASSETTEOBJECT_TOKEN	1266
MODIFY_MEREVENTLISTENER_TOKEN	1269
MODIFY_PAYSERVER_TOKEN	1251
MODIFY_PAYSYSTEM_TOKEN	1257
MODIFY_SNMEVENTLISTENER_TOKEN	1307
MODIFY_SYSTEMCASSETTEOBJECT_TOKEN	1263
MODIFY_USERSTATUS_TOKEN	1279
RECEIVEPAYMENT_TOKEN	1200
REFUND_TOKEN	1203
REFUNDREVERSAL_TOKEN	1206
SET_USERACCESSRIGHTS_TOKEN	1277

Query command requests extend the `com.ibm.etill.framework.xdm.QueryRequest` base class. The query request class hierarchy is as follows:

```

java.lang.Object
|
+--com.ibm.etill.framework.xdm.QueryRequest
    |
    +--com.ibm.etill.framework.xdm.AccountQueryRequest
    +--com.ibm.etill.framework.xdm.CassetteQueryRequest
    +--com.ibm.etill.framework.xdm.FinancialObjectQueryRequest
        |
        +--com.ibm.etill.framework.xdm.BatchQueryRequest
        +--com.ibm.etill.framework.xdm.TransactionQueryRequest
            |
            +--com.ibm.etill.framework.xdm.CreditQueryRequest
            +--com.ibm.etill.framework.xdm.OrderQueryRequest
            +--com.ibm.etill.framework.xdm.PaymentQueryRequest
    +--com.ibm.etill.framework.xdm.PaymentSystemQueryRequest
  
```

Protecting sensitive data

If the payment cassette you are writing contains sensitive data, you should be aware that you can configure a Payment Servlet initialization parameter (a JVM system parameter), `wpm.MinSensitiveAccessRole`, to restrict users from accessing the data in query command results. Sensitive data could be credit card numbers, buyer bank account numbers, verification codes or data, or anything you would consider to be sensitive enough to warrant encryption before storage in the WebSphere Commerce Payments database.

Through the WebSphere Commerce Configuration Manager, you can define a minimum sensitive access role by setting the Minimum Access Role field for the Payments instance. This field defines what minimum role a user should have to be allowed access to the data returned when query commands are entered. Role values are clerk, supervisor, madmin (Merchant Administrator), psadmin (Payments Administrator), or none. If a parameter value is not specified, a value of clerk is assumed, allowing all users to see sensitive data.

For each query command, the framework verifies the user's role against the minimum role specified by this parameter, and sets an indicator in the QueryRequest object to indicate whether sensitive data should be returned in full view or if it should be masked out with asterisks. The following methods are available to cassette writers for supporting the wpm.MinSensitiveAccessRole property: QueryRequest.getShowSensitiveData() and QueryResponse.maskSensitiveData(). These methods enable you to check the value of this indicator and also to mask sensitive data in a standardized way.

If you use this masking function, note the following:

- As a cassette writer, you need to discern what data is sensitive. Currently, the WebSphere Commerce Payments framework does not maintain any sensitive data that can be returned through a query command. Typically, sensitive data is the same set of data that a cassette encrypts before storing it to the WebSphere Commerce Payments database.
- The following methods are available to support the masking function:
 - For the `com.ibm.etill.framework.xdm.QueryRequest` class, you can use the `getShowSensitiveData` method. This method indicates whether the query request should return the cassette's sensitive data to the user or hide it by masking with asterisks.
 - For the `com.ibm.etill.framework.xdm.QueryResponse` class, you can use the `maskSensitiveData` method. This method masks all or part of a string of sensitive data up to 80 characters long with asterisks so that it can be safely returned in a query command response. When calling the method, specify the string containing the data to be masked, and an integer value to indicate how many characters to show and whether they should be leading or trailing characters. For example:
 - A negative integer value of $-n$ causes the first n characters of the actual value to be inserted in the string. (For instance, if the sensitive data string is 1111222233334444, a value of -3 means that the first three characters are included and all remaining characters are masked. The resulting string is 111*****.)
 - A positive value of n causes the last n characters of the actual value to be included in the string. Using the previous string example, a value of 3 means that the resulting string would be: *****444 .
 - A value of 0 indicates the entire value should be replaced with asterisks. In this example, the resulting string is ***** .
 - If the value is greater than the length of the input string, the entire string is returned unmasked.

Refer to the Javadoc for more information about using these classes to define the methods used by your cassette:

- `com.ibm.etill.framework.xdm.QueryRequest`
- `com.ibm.etill.framework.xdm.QueryResponse`

Cassettes and command processing

This section describes:

- The facilities available to cassettes for tailoring the command interface for their own needs
- The impact of customized interfaces on merchant software,
- The strategies and tactics for minimizing the impact on merchant software.

Protocol data: Cassette-specific command parameters

As mentioned above, given the wide variety of existing and potential payment protocols, it is certain that cassettes will need to define their own command parameters in certain cases. To address this need, the framework allows cassettes to define protocol data parameters. Protocol data parameters are keyword/value pairs, just like the framework command parameters. Protocol data keywords must be in the form:

\$keyword

The maximum keyword length (excluding the '\$' symbol) is 254 characters and all of the characters must be valid ISO 8859-1 characters. The legal parameter values vary with each parameter - this is part of the parameter's definition and is the responsibility of the person that creates the parameter.

Cassette writers should define new protocol data parameters for a given Payment command only after they have done the following:

- Determined that none of the framework command parameters contain the data that their payment protocol requires to process the command
- Examined the list of common protocol data parameters in the table below and determined that none of those parameters will suit their needs
- Checked with IBM's payment cassette development support group so the new parameter can be evaluated for inclusion to the list of common protocol data parameters.

Using this process, cassette writers will ensure that the merchant software view of their cassette is as consistent as it can be with all other payment cassettes. In order to determine if the list of common protocol data defined by the framework is sufficient for your cassette's needs, the following table provides a list of all common protocol data defined by the framework. It is important that you follow the guidelines in the table so that merchant server software can expect consistent behavior across all cassettes for these common pieces of data. The table contains:

- Protocol data constant (as defined in `com.ibm.etill.framework.payapi.PaymentAPIConstants`).
- The actual string value of the protocol data.
- The associated return code constant that should be returned by the cassette (as defined in `com.ibm.etill.framework.payapi.FrameworkReturnCodes`) in the case of an error.
- Which ValidatorItem to use (see "Parameter Validation" "Parameter validation" on page 96 for details)
- If your cassette stores the protocol data value in the database, what the database type should be. An "*" in the column indicates that the data is sensitive and should be encrypted.

- The corresponding XML attribute for the protocol data (see "Exported Data Model: The user's view of WebSphere Commerce Payments objects" "Exported Data Model: The user's view of WebSphere Commerce Payments objects" on page 22 for details).

Table 2. Protocol Data Chart

Protocol data constant	Protocol data value	Return code	Validation Item	Database Type	XML Attribute
PD_AUTHCODE	\$AUTHCODE	RC_CASSETTE_AUTHCODE	StringValidator	VARCHAR	authCode
PD_AUXILLARY1	\$AUXILLARY1	RC_CASSETTE_AUXILLARY1	StringValidator	VARCHAR	auxillary1
PD_AUXILLARY2	\$AUXILLARY2	RC_CASSETTE_AUXILLARY2	StringValidator	VARCHAR	auxillary2
PD_AVS_CITY	\$AVS_CITY	RC_CASSETTE_AVS_CITY	StringValidator	VARCHAR	avsCity
PD_AVS_COUNTRYCODE	\$AVS_COUNTRYCODE	RC_CASSETTE_AVS_COUNTRYCODE	IntegerValidator	INTEGER	avsCountryCode
PD_AVS_LOCATIONID	\$AVS_LOCATIONID	RC_CASSETTE_AVS_LOCATIONID	StringValidator	VARCHAR	avsLocationID
PD_AVS_POSTALCODE	\$AVS_POSTALCODE	RC_CASSETTE_AVS_POSTALCODE	StringValidator	VARCHAR	avsPostalCode
PD_AVS_STATE_PROVINCE	\$AVS_STATE_PROVINCE	RC_CASSETTE_AVS_STATE_PROVINCE	StringValidator	VARCHAR	avsStateProvince
PD_AVS_STREETADDRESS	\$AVS_STREETADDRESS	RC_CASSETTE_AVS_STREETADDRESS	StringValidator	VARCHAR	avsStreetAddress
PD_AVS_CODE	\$AVS_CODE	RC_CASSETTE_AVS_CODE	StringValidator	VARCHAR	avsCode
PD_BATCHCLOSETIME	\$BATCHCLOSETIME	RC_CASSETTE_BATCHCLOSETIME	IntegerValidator	SMALLINT	batchCloseTime
PD_BILL_CITY	\$BILL_CITY	RC_CASSETTE_BILL_CITY	StringValidator	VARCHAR	billCity
PD_BILL_COUNTRYCODE	\$BILL_COUNTRYCODE	RC_CASSETTE_BILL_COUNTRYCODE	IntegerValidator	INTEGER	billCountryCode
PD_BILL_POSTALCODE	\$BILL_POSTALCODE	RC_CASSETTE_BILL_POSTALCODE	StringValidator	VARCHAR	billPostalCode
PD_BILL_STATEPROVINCE	\$BILL_STATEPROVINCE	RC_CASSETTE_BILL_STATEPROVINCE	StringValidator	VARCHAR	billStateProvince
PD_BILL_STREETADDRESS	\$BILL_STREETADDRESS	RC_CASSETTE_BILL_STREETADDRESS	StringValidator	VARCHAR	billStreetAddress
PD_BRAND	\$BRAND	RC_CASSETTE_BRAND	RestrictedStringValidator	VARCHAR	brand
PD_CARDHOLDERNAME	\$CARDHOLDERNAME	RC_CASSETTE_CARDHOLDERNAME	StringValidator	VARCHAR	cardHolderName
PD_CARDVERIFYCODE	\$CARDVERIFYCODE	RC_CASSETTE_CARDVERIFYCODE	NumericStringValidator	VARCHAR*	cardVerifyCode
PD_CITY	\$CITY	RC_CASSETTE_CITY	StringValidator	VARCHAR	city
PD_COUNTRYCODE	\$COUNTRYCODE	RC_CASSETTE_COUNTRYCODE	IntegerValidator	INTEGER	countrycode
PD_CURRENCY	\$CURRENCY	RC_CASSETTE_CURRENCY	NumericStringValidator	SMALLINT	currency
PD_DECLINECODE	\$DECLINECODE	RC_CASSETTE_DECLINECODE	StringValidator	VARCHAR	declineCode
PD_DECLINEREASON	\$DECLINEREASON	RC_CASSETTE_DECLINECODE	StringValidator	VARCHAR	declineReason
PD_EXPIRY	\$EXPIRY	RC_CASSETTE_EXPIRY	NumericStringValidator	VARCHAR*	expiry
PD_FIBATCHID	\$FIBATCHID	RC_CASSETTE_FIBATCHID	NumericTokenValidator	INTEGER	fiBatchId
PD_ITEM_COMMODITYCODE	\$ITEM.COMMODITYCODE	RC_CASSETTE_ITEM_COMMODITYCODE	StringValidator	VARCHAR	commodityCode.n
PD_ITEM_PRODUCTCODE	\$ITEM.PRODUCTCODE	RC_CASSETTE_ITEM_PRODUCTCODE	StringValidator	VARCHAR	productCode.n
PD_ITEM_DESCRIPTOR	\$ITEM.DESRIPTOR	RC_CASSETTE_ITEM_DESCRIPTOR	StringValidator	VARCHAR	descriptor.n
PD_ITEM_QUANTITY	\$ITEM.QUANTITY	RC_CASSETTE_ITEM_QUANTITY	StringValidator	VARCHAR	quantity.n
PD_ITEM_SKU	\$ITEM.SKU	RC_CASSETTE_ITEM_SKU	StringValidator	VARCHAR	SKU.n
PD_ITEM_UNITCOST	\$ITEM.UNITCOST	RC_CASSETTE_ITEM_UNITCOST	StringValidator	VARCHAR	unitCost.n
PD_ITEM_UNITOFMEASURE	\$ITEM.UNITOFMEASURE	RC_CASSETTE_ITEM_UNITOFMEASURE	StringValidator	VARCHAR	unitOfMeasure.n
PD_ITEM_NETCOST	\$ITEM.NETCOST	RC_CASSETTE_ITEM_NETCOST	StringValidator	VARCHAR	netCost.n
PD_ITEM_DISCOUNTAMOUNT	\$ITEM.DISCOUNTAMOUNT	RC_CASSETTE_ITEM_DISCOUNTAMOUNT	StringValidator	VARCHAR	discountAmount.n
PD_ITEM_DISCOUNTINDICATOR	\$ITEM.DISCOUNTINDICATOR	RC_CASSETTE_ITEM_DISCOUNTINDICATOR	StringValidator	VARCHAR	discountIndicator.n
PD_ITEM_NATIONALTAXAMOUNT	\$ITEM.NATIONALTAXAMOUNT	RC_CASSETTE_ITEM_NATIONALTAXAMOUNT	StringValidator	VARCHAR	nationalTaxAmount.n
PD_ITEM_NATIONALTAXRATE	\$ITEM.NATIONALTAXRATE	RC_CASSETTE_ITEM_NATIONALTAXRATE	StringValidator	VARCHAR	nationalTaxRate.n
PD_ITEM_NATIONALTAXTYPE	\$ITEM.NATIONALTAXTYPE	RC_CASSETTE_ITEM_NATIONALTAXTYPE	StringValidator	VARCHAR	nationalTaxType.n
PD_ITEM_LOCALTAXAMOUNT	\$ITEM.LOCALTAXAMOUNT	RC_CASSETTE_ITEM_LOCALTAXAMOUNT	StringValidator	VARCHAR	localTaxAmount.n
PD_ITEM_LOCALTAXRATE	\$ITEM.LOCALTAXRATE	RC_CASSETTE_LOCALTAXRATE	StringValidator	VARCHAR	localTaxRate.n
PD_ITEM_OTHERTAXAMOUNT	\$ITEM.OTHERTAXAMOUNT	RC_CASSETTE_ITEM_OTHERTAXAMOUNT	StringValidator	VARCHAR	otherTaxAmount.n
PD_ITEM_TOTALCOST	\$ITEM.TOTALCOST	RC_CASSETTE_ITEM_TOTALCOST	StringValidator	VARCHAR	totalCost.n
PD_MAXBATCHSIZE	\$MAXBATCHSIZE	RC_CASSETTE_MAXBATCHSIZE	IntegerValidator	INTEGER	maxBatchSize
PD_METHOD	\$METHOD	RC_CASSETTE_METHOD	StringValidator	VARCHAR	method
PD_PAN	\$PAN	RC_CASSETTE_PAN	NumericStringValidator	VARCHAR*	pan
PD_PCARD_SHIPPINGAMOUNT	\$PCARD.SHIPPINGAMOUNT	RC_CASSETTE_PCARD_SHIPPINGAMOUNT	StringValidator	VARCHAR	shippingAmount
PD_PCARD_DUTYAMOUNT	\$PCARD.DUTYAMOUNT	RC_CASSETTE_PCARD_DUTYAMOUNT	StringValidator	VARCHAR	dutyAmount
PD_PCARD_DUTYREFERENCE	\$PCARD.DUTYREFERENCE	RC_CASSETTE_PCARD_DUTYREFERENCE	StringValidator	VARCHAR	dutyReference
PD_PCARD_NATIONAL TAXAMOUNT	\$PCARD.NATIONALTAXAMOUNT	RC_CASSETTE_PCARD_NATIONALTAXAMOUNT	StringValidator	VARCHAR	nationalTaxAmount
PD_PCARD_NATIONALTAXRATE	\$PCARD.NATIONALTAXRATE	RC_CASSETTE_PCARD_NATIONALTAXRATE	StringValidator	VARCHAR	nationalTaxRate
PD_PCARD_LOCALTAXAMOUNT	\$PCARD.LOCALTAXAMOUNT	RC_CASSETTE_PCARD_LOCALTAXAMOUNT	StringValidator	VARCHAR	localTaxAmount
PD_PCARD_OTHERTAXAMOUNT	\$PCARD.OTHERTAXAMOUNT	RC_CASSETTE_PCARD_OTHERTAXAMOUNT	StringValidator	VARCHAR	otherTaxAmount
PD_PCARD_TOTALTAXAMOUNT	\$PCARD.TOTALTAXAMOUNT	RC_CASSETTE_PCARD_TOTALTAXAMOUNT	StringValidator	VARCHAR	totalTaxAmount
PD_PCARD_MERCHANTTAXID	\$PCARD.MERCHANTTAXID	RC_CASSETTE_PCARD_MERCHANTTAXID	StringValidator	VARCHAR	merchantTaxId
PD_PCARD_ALTERNATETAXID	\$PCARD.ALTERNATETAXID	RC_CASSETTE_PCARD_ALTERNATETAXID	StringValidator	VARCHAR	alternateTaxId
PD_PCARD_TAXEXEMPTINDICATOR	\$PCARD.TAXEXEMPTINDICATOR	RC_CASSETTE_PCARD_TAXEXEMPTINDICATOR	StringValidator	VARCHAR	taxExemptIndicator
PD_PCARD_MERCHANTDUTY TARIFFREFERENCE	\$PCARD.MERCHANTDUTY TARIFFREFERENCE	RC_CASSETTE_PCARD_MERCHANTDUTYTARIFFREFERENCE	StringValidator	VARCHAR	merchantDutyTariffReference
PD_PCARD_CUSTOMERDUTYTARIFF REFERENCE	\$PCARD.CUSTOMERDUTYTARIFF REFERENCE	RC_CASSETTE_PCARD_CUSTOMERDUTYTARIFFREFERENCE	StringValidator	VARCHAR	customerDutyTariffReference
PD_PCARD_SUMMARYCOMMODITYCODE	\$PCARD.SUMMARYCOMMODITY CODE	RC_CASSETTE_PCARD_SUMMARYCOMMODITYCODE	StringValidator	VARCHAR	summaryCommodityCode

Table 2. Protocol Data Chart (continued)

Protocol data constant	Protocol data value	Return code	Validation Item	Database Type	XML Attribute
PD_PCARD_MERCHANTTYPE	\$PCARD.MERCHANTTYPE	RC_CASSETTE_PCARD_MERCHANTTYPE	StringValidator	VARCHAR	merchantType
PD_PCARD_MERCHANTCOUNTRYCODE	\$PCARD.MERCHANTCOUNTRYCODE	RC_CASSETTE_PCARD_MERCHANTCOUNTRYCODE	StringValidator	VARCHAR	merchantCountryCode
PD_PCARD_MERCHANTCITYCODE	\$PCARD.MERCHANTCITYCODE	RC_CASSETTE_PCARD_MERCHANTCITYCODE	StringValidator	VARCHAR	merchantCityCode
PD_PCARD_MERCHANTSTATEPROVINCE	\$PCARD.MERCHANTSTATEPROVINCE	RC_CASSETTE_PCARD_MERCHANTSTATEPROVINCE	StringValidator	VARCHAR	merchantStateProvince
PD_PCARD_MERCHANTPOSTALCODE	\$PCARD.MERCHANTPOSTALCODE	RC_CASSETTE_PCARD_MERCHANTPOSTALCODE	StringValidator	VARCHAR	merchantPostalCode
PD_PCARD_MERCHANTLOCATIONID	\$PCARD.MERCHANTLOCATIONID	RC_CASSETTE_PCARD_MERCHANTLOCATIONID	StringValidator	VARCHAR	merchantLocationId
PD_PCARD_MERCHANTNAME	\$PCARD.MERCHANTNAME	RC_CASSETTE_PCARD_MERCHANTNAME	StringValidator	VARCHAR	merchantName
PD_PCARD_SHIPFROMCOUNTRYCODE	\$PCARD.SHIPFROMCOUNTRYCODE	RC_CASSETTE_PCARD_SHIPFROMCOUNTRYCODE	StringValidator	VARCHAR	shipFromCountryCode
PD_PCARD_SHIPFROMCITYCODE	\$PCARD.SHIPFROMCITYCODE	RC_CASSETTE_PCARD_SHIPFROMCITYCODE	StringValidator	VARCHAR	shipFromCityCode
PD_PCARD_SHIPFROMSTATEPROVINCE	\$PCARD.SHIPFROMSTATEPROVINCE	RC_CASSETTE_PCARD_SHIPFROMSTATEPROVINCE	StringValidator	VARCHAR	shipFromStateProvince
PD_PCARD_SHIPFROMPOSTALCODE	\$PCARD.SHIPFROMPOSTALCODE	RC_CASSETTE_PCARD_SHIPFROMPOSTALCODE	StringValidator	VARCHAR	shipFromPostalCode
PD_PCARD_SHIPFROMLOCATIONID	\$PCARD.SHIPFROMLOCATIONID	RC_CASSETTE_PCARD_SHIPFROMLOCATIONID	StringValidator	VARCHAR	shipFromLocationId
PD_PCARD_SHIPTOCOUNTRYCODE	\$PCARD.SHIPTOCOUNTRYCODE	RC_CASSETTE_PCARD_SHIPTOCOUNTRYCODE	StringValidator	VARCHAR	shipToCountryCode
PD_PCARD_SHIPTOCITYCODE	\$PCARD.SHIPTOCITYCODE	RC_CASSETTE_PCARD_SHIPTOCITYCODE	StringValidator	VARCHAR	shipToCityCode
PD_PCARD_SHIPTOSTATEPROVINCE	\$PCARD.SHIPTOSTATEPROVINCE	RC_CASSETTE_PCARD_SHIPTOSTATEPROVINCE	StringValidator	VARCHAR	shipToStateProvince
PD_PCARD_SHIPTOPOSTALCODE	\$PCARD.SHIPTOPOSTALCODE	RC_CASSETTE_SHIPTOPOSTALCODE	StringValidator	VARCHAR	shipToPostalCode
PD_PCARD_SHIPTOLOCATIONID	\$PCARD.SHIPTOLOCATIONID	RC_CASSETTE_SHIPTOLOCATIONID	StringValidator	VARCHAR	shipToLocationId
PD_PCARD_MERCHANTORDERNUMBER	\$PCARD.MERCHANTORDERNUMBER	RC_CASSETTE_MERCHANTORDERNUMBER	StringValidator	VARCHAR	merchantOrderNumber
PD_PCARD_CUSTOMERREFERENCE NUMBER	\$PCARD.CUSTOMERREFERENCE NUMBER	RC_CASSETTE_PCARD_CUSTOMERREFERENCENUMBER	StringValidator	VARCHAR	customReferenceNumber
PD_PCARD_ORDERSUMMARY	\$PCARD.ORDERSUMMARY	RC_CASSETTE_PCARD_ORDERSUMMARY	StringValidator	VARCHAR	orderSummary
PD_PCARD_CUSTOMERSERVICE PHONE	\$PCARD.CUSTOMERSERVICE PHONE	RC_CASSETTE_PCARD_CUSTOMERSERVICEPHONE	StringValidator	VARCHAR	customerServicePhone
PD_PCARD_DISCOUNTAMOUNT	\$PCARD.DISCOUNTAMOUNT	RC_CASSETTE_PCARD_DISCOUNTAMOUNT	StringValidator	VARCHAR	discountAmount
PD_PCARD_SHIPPINGNATIONALTAXRATE	\$PCARD.SHIPPINGNATIONAL TAXRATE	RC_CASSETTE_PCARD_SHIPPINGNATIONALTAXRATE	StringValidator	VARCHAR	shippingNationalTaxRate
PD_PCARD_SHIPPINGNATIONAL TAXAMOUNT	\$PCARD.SHIPPINGNATIONAL TAXAMOUNT	RC_CASSETTE_PCARD_SHIPPINGNATIONALTAXAMOUNT	StringValidator	VARCHAR	shippingNationalTaxAmount
PD_PCARD_NATIONALTAX INVOICEREFERENCE	\$PCARD.NATIONALTAX INVOICEREFERENCE	RC_CASSETTE_PCARD_NATIONALTAXINVOICEREFERENCE	StringValidator	VARCHAR	nationalTaxInvoiceReference
PD_PCARD_PRINTCUSTOMER SERVICEPHONENUMBER	\$PCARD.PRINTCUSTOMER SERVICEPHONENUMBER	RC_CASSETTE_PCARD_PRINTCUSTOMERSERVICEPHONENUMBER	StringValidator	VARCHAR	printCustomerServicePhone Number
PD_POSTALCODE	\$POSTALCODE	RC_CASSETTE_POSTALCODE	StringValidator	VARCHAR	postalCode
PD_SHIP_CITY	\$SHIP_CITY	RC_CASSETTE_SHIP_CITY	StringValidator	VARCHAR	shipCity
PD_SHIP_COUNTRYCODE	\$SHIP_COUNTRYCODE	RC_CASSETTE_SHIP_COUNTRYCODE	IntegerValidator	INTEGER	shipCountryCode
PD_SHIP_POSTALCODE	\$SHIP_POSTALCODE	RC_CASSETTE_SHIP_POSTALCODE	StringValidator	VARCHAR	shipPostalCode
PD_SHIP_STATEPROVINCE	\$SHIP_STATEPROVINCE	RC_CASSETTE_SHIP_STATEPROVINCE	StringValidator	VARCHAR	shipStateProvince
PD_SHIP_STREETADDRESS	\$SHIP_STREETADDRESS	RC_CASSETTE_SHIP_STREETADDRESS	StringValidator	VARCHAR	shipStreetAddress
PD_STREETADDRESS	\$STREETADDRESS	RC_CASSETTE_STREETADDRESS	StringValidator	VARCHAR	streetAddress
PD_STATEPROVINCE	\$STATEPROVINCE	RC_CASSETTE_STATEPROVINCE	StringValidator	VARCHAR	stateProvince

(* = data is sensitive and should be encrypted by your cassette if stored in the database)

If you do end up defining protocol data parameters, here are some guidelines to follow to ensure consistency for the merchant software interface:

- Keyword names should be descriptive. For example, \$USERACCOUNTINDEX, instead of \$UAX. In some cases, where acronyms are well-known in the payment community, the acronym may be suitable (for example, \$PAN, for personal account number).

- When multiple related parameters are being defined, they should all begin with a common prefix (for example, \$AVS.xxx for all of the Address Verification Service-related parameters). This concept can be carried into multiple levels (for example, \$SD.CCARD.xxx for Sale Detail, and so on).
- As already illustrated, when multiple qualifiers are used within a parameter name, the qualifiers should be separated by '.' (dot) characters.
- A single keyword should represent a single value. Don't pass complex structures or sets of values as a single parameter value if at all possible. Doing so can cause compatibility problems for other cassette writers.
- Previous versions of this document recommended that multiple values (i.e., a collection of values) required for a common parameter type should be indicated by appending an index to the keyword and separate the two with a '.' (dot) character. For example, \$SD.CCARD.ITEM.QUANTITY.1. This process is acceptable; however, you will want to consider the following:
 1. If the framework has already defined a standard protocol data parameter keyword suitable for the collection, choose that parameter keyword for your cassette's collection of values. If not, define a new parameter keyword. If you believe that the new keyword may be valuable as a standard for other cassettes, propose it as a standard protocol data parameter keyword to be supplied by the framework. If the collection will be passed as elements in an Extensible Markup Language (XML) document, the parameter name should begin with \$XML followed by a period followed by the XML document type.
 2. As much as possible, choose a universal format that allows the merchant software to pass the collection of values as a string of characters.
 - If there is an industry standard format for the string, choose that format.

In the *Cassette Kit Programmer's Guide, version 2.2*, we recommended that the merchant supply line item detail in an XML document using tags approved by an e-commerce standards committee. We suggested two alternatives: Common Business Language (xCBL) proposed by CommerceOne (see <http://www.commerceone.com/xml/index.html> for specifications) and Commerce Extensible Markup Language (cXML) (see <http://www.cxml.org> for endorsements and specifications). Due to compatibility conflicts with these standards, we are no longer making this recommendation.
 - If there is no industry standard, consider defining an XML document to contain the collection. If justified, build a document type definition (DTD) that defines the XML tags that the merchant will use.
 - If a new XML document type is not justified, use a simple format. For example, you could use comma separated character strings and, if necessary, semicolons to separate collections within collections. If you find yourself defining a format with keyword value pairs, reconsider using an XML format. Typically, XML provides a more robust definition and an XML document will be easier for the cassette to parse.
 3. To facilitate automatic parameter value validation and conversion, implement a subclass that extends the StringValidator class. Use the parent's methods to convert the collection of values into a Java String and then, if necessary, parse the string to validate and convert the values in the collection. If the string is an XML document, you can use the XML parser that is provided by the framework.
- If you are writing your cassette to process purchasing card data, you can take advantage of the WebSphere Commerce Payments framework services available to process the repeating line item detail associated with purchasing card data (for example, \$ITEM.QUANTITY.1, \$ITEM.QUANTITY.2, \$ITEM.QUANTITY.3,

and so on). Refer to “Purchasing card support” on page 112 for a description of purchasing cards and more information about these framework services.

Remember, the framework is going to use a `com.ibm.etill.payapi.ParameterTable` (Java hash table) to pass the result to the cassette and the protocol data parameter keyword will be the key. Your new validation class will need to return a single object that contains all the values in the collection. Consider returning a hash table or a vector. If neither are suitable, implement a new class to contain the collection with methods that simplify the cassette’s logic when it needs to process the collection.

Although the framework will allow protocol data parameters on any API command, you should try to limit them to the Administrative commands and the financial commands that create Order objects (that is, the `AcceptPayment` and `ReceivePayment` commands). Since the vast majority of data items required by a specific payment protocol can easily be considered attributes of the order itself, it makes sense to pass those attributes into WebSphere Commerce Payments when the Order is created. The only time you should define a protocol data parameter for a command other than those mentioned here is when this statement does not apply to the parameter.

Ignore or reject?

Cassette writers will find themselves faced with the question “ignore or reject?” at several times throughout the design and development process. Use the following answers as guidelines on how to answer that question. These answers are intended to encourage common semantics through the WebSphere Commerce Payments API, regardless of the underlying cassette, which fosters the ability to write payment protocol-independent merchant software.

Q. When an unsupported command is received, should my cassette ignore it (return “success”) or reject it (return “command not supported”)?

A. If a command is not directly supported by your payment protocol but can be simulated, then accept it. For example, you may be able to simulate an `ApproveReversal` by voiding an authorization and making a new authorization. If the unsupported command cannot be simulated, then reject it with `PRC_COMMAND_NOT_SUPPORTED`, `RC_NONE` instead of ignoring its existence. Merchant software writers should expect and tolerate this return code pair as a likely possibility and handle it accordingly.

Q. Assume that my payment protocol automatically deposits and closes payments as part of the approval process. Should my cassette require that the `DEPOSITFLAG=1` parameter be specified on the `Approve` command, or should it ignore the flag completely?

A. Your cassette should require such parameters even when a behavior is implied by the payment protocol. For more details on this, see “Supporting framework command parameters” on page 37.

Q. What should my cassette do if it receives an unrecognized protocol data parameter?

A. Your cassette should ignore these. If cassettes were to reject unrecognized protocol data parameters, the merchant software would have to ensure that every protocol data parameter passed on every payment command is supported by the cassette that will eventually handle the request (in other words, the merchant software writer would have to be very sensitive to the underlying payment protocol). By ignoring unrecognized protocol data parameters, cassettes allow

merchant software to code their payment commands once with all relevant parameters. The cassette can then process only the subset of those parameters that it understands.

Supporting framework command parameters

Framework command parameters that require cassette-specific logic are described here. To preserve a predictable and consistent merchant programming view, cassette writers should support these parameters as defined by the framework's programming documentation:

Command	Parameter	Description
AcceptPayment	APPROVEFLAG	If requested, approval should be performed after the Order object is successfully created. If the approval step fails, the Order object should remain in existence and a primary return code of PRC_AUTOAPPROVE_FAILED should be returned with a suitable secondary return code. If the APPROVEFLAG equals 0 or 2, an approval should not be performed.
	DEPOSITFLAG	May only be requested if AUTOAPPROVE was also requested. If requested, then a deposit should be performed after successful approval. If the deposit step fails, the Order and Payment objects should remain in existence and a primary return code of PRC_AUTODEPOSIT_FAILED should be returned with a suitable secondary return code. Note that even if your payment protocol already automatically captures funds when approvals are executed, your cassette should still require this parameter. In such cases, return codes PRC_PARAMETER_NOT_FOUND, RC_DEPOSITFLAG should be returned if AUTOAPPROVE is specified without AUTODEPOSIT.
Approve	DEPOSITFLAG	If requested, then a deposit should be performed after successful approval. If the deposit step fails, the Order and Payment objects should remain in existence and a primary return code of PRC_AUTODEPOSIT_FAILED should be returned with a suitable secondary return code. Note that even if your payment protocol already automatically captures funds when approvals are executed, your cassette should still require this parameter. In such cases, return codes PRC_PARAMETER_NOT_FOUND, RC_DEPOSITFLAG should be returned if AUTODEPOSIT is not specified.
CloseOrder	DELETEORDER	If specified, the cassette should delete the Order and all of its Payments and Credits (which can be done via call to Supervisor.pruneOrder) after the Order is successfully closed.
CancelOrder	DELETEORDER	If specified, the cassette should delete the Order and all of its Payments and Credits (which can be done via call to Supervisor.pruneOrder) after the Order is successfully canceled.
ReceivePayment	APPROVEFLAG	Same as for AcceptPayment
	DEPOSITFLAG	Same as for AcceptPayment

Error handling

WebSphere Commerce Payments uses the conventions and facilities described here for reporting errors. Cassette writers should use these as described:

General runtime errors - `ETillAbortOperation`

The most common type of error case will be non-fatal errors detected while a command is being processed. Non-fatal means that the requested operation cannot complete successfully, but WebSphere Commerce Payments and the cassette are still able to function normally and continue to process new commands. From the merchant software point of view, such errors will result in a command response that contains a primary and secondary return code that indicate the type of failure that took place. To generate these codes and the associated response, cassettes must throw a `com.ibm.etill.framework.payapi.ETillAbortOperation` exception. This exception object allows you to specify a primary and secondary return code. If it is thrown while a command is being processed, the framework will automatically generate a command response that contains the specified error codes. Cassettes should only use the primary return codes defined in `com.ibm.etill.framework.payapi.FrameworkReturnCodes`. Where there is no primary return code that directly relates to the error, the cassette should use `PRC_CASSETTE_ERROR` with an appropriate secondary return code. *Cassettes should never define their own primary return code values!*

Additionally, a large number of common secondary return codes are defined in `com.ibm.etill.framework.payapi.FrameworkReturnCodes`. Use these codes before defining your own. If a suitable code is not found there, follow the procedure described above for protocol data.

Again, IBM is very interested in maintaining as much commonality across error codes as possible. If, after following the procedures above, you define your own cassette-specific return codes, use a numeric value greater than 10000. The range between 0 and 9999 is reserved for IBM-defined return codes, which will include the common cassette return codes that are created over time. Also, you must describe any new return codes that you define in your cassette documentation.

Handling unsupported commands

As previously described, cassette writers must make conscious decisions about which commands their cassette will support. To provide a consistent interface for merchant software, cassettes should always respond to unsupported commands by throwing an `ETillAbortOperation` exception with:

- Primary return code = `PRC_CASSETTE_ERROR` and
- Secondary return code = `RC_COMMAND_NOT_SUPPORTED`

Both of these codes are defined in `com.ibm.etill.framework.payapi.FrameworkReturnCodes`.

Fatal errors

In extreme cases where errors occur that make it impossible for the cassette to reliably continue operation, the cassette should throw a `com.ibm.etill.framework.log.ETillCassetteException` exception. This exception will cause the framework to stop and disable the cassette.

Errors while processing protocol messages

Each payment protocol that supports its own protocol messages will most likely have its own mechanism for reporting errors to the entity with

which protocol messages are being exchanged. Therefore, the framework does not attempt to solve this problem other than providing the tracing and logging facilities through which cassettes can record error events as they occur. The use of these facilities, of course, is left to the discretion of the cassette writer.

Cassette-specific commands: `CassetteControl`

The `CassetteControl` command is a more extreme case of defining cassette-specific behavior. This is more extreme because each use of this command effectively defines a new protocol-specific command with which merchant software must contend. While there will be cases where `CassetteControl` is required, its use should only be a last resort. Again, if your payment protocol requires a new command, please consult with IBM's payment cassette development support group to discuss the requirement and agree on an approach that will minimize the impact to merchant software.

Administrative API commands

The administrative API commands create, modify and delete the administrative objects that represent the main participants in financial transactions (such as merchants and accounts) as well as major components of WebSphere Commerce Payments itself (such as cassettes and event listeners).

There are a few basic differences between the way the cassette is expected to handle administrative and payment commands:

- All administrative commands are represented by an instance of the `AdminRequest` class. That is, this class is not subclassed for the various different types of administrative commands. Every payment command, on the other hand, has its own class that is derived from `APIRequest`.
- For payment commands, the cassette's service method is called by the framework only once for each payment command. For administrative commands, the service method is called twice but only for non-merchant cassette objects and non-system cassette objects:
 - The first call is a request for the cassette to validate any protocol data parameters (in this case, the request is considered to be in "validation mode"). Note that when validation mode calls are made to the service method, the associated `AdminRequest` object does not yet exist.
 - The second call is a request for the cassette to actually process the command.

Both calls are made using the same `AdminRequest` object: the cassette determines whether the request is in validation mode by calling the `validationMode()` method on the `AdminRequest` object.

For merchant cassette objects and system cassette objects, there is no call for validation. There is only a call to process the command.

- The cassette is asked to participate in the processing of every payment command, whereas it is only asked to participate for the subset of administrative commands for which cassette extensions are allowed (that is, `CassetteAdmin`, `MerchantAdmin`, `AccountAdmin` and `PaySystemAdmin` objects) and for cassette-defined primary administrative objects (that is, merchant cassette objects and system cassette objects).

Administrative commands reflect the Administrative Object hierarchy. The Administrative object hierarchy determines what the administrative commands must do. For example, if an object in the middle of the hierarchy is stopped, all

objects beneath that object must also be stopped. The administrative commands invoke internal functions, as shown in the subsequent diagrams.

Because of the well-defined hierarchy between the various framework administrative objects the internal operation of the administrative commands has a very similar hierarchy. For example, AccountAdmin objects are "contained in" a PaySystemAdmin object. Therefore, if you start (enable) a payment system, the command must also start all of that payment system's accounts. To clearly and accurately reflect these relationships, the administrative command flows include two different types of diagrams:

- Internal sequence diagrams, which illustrate the internal flows required to perform a specific action (start, stop or delete). These sequences may be invoked from multiple places within the framework.
- API sequence diagrams, which show how a particular command is received and routed within the Payment Servlet. These flows will eventually invoke one or more of the internal sequences.

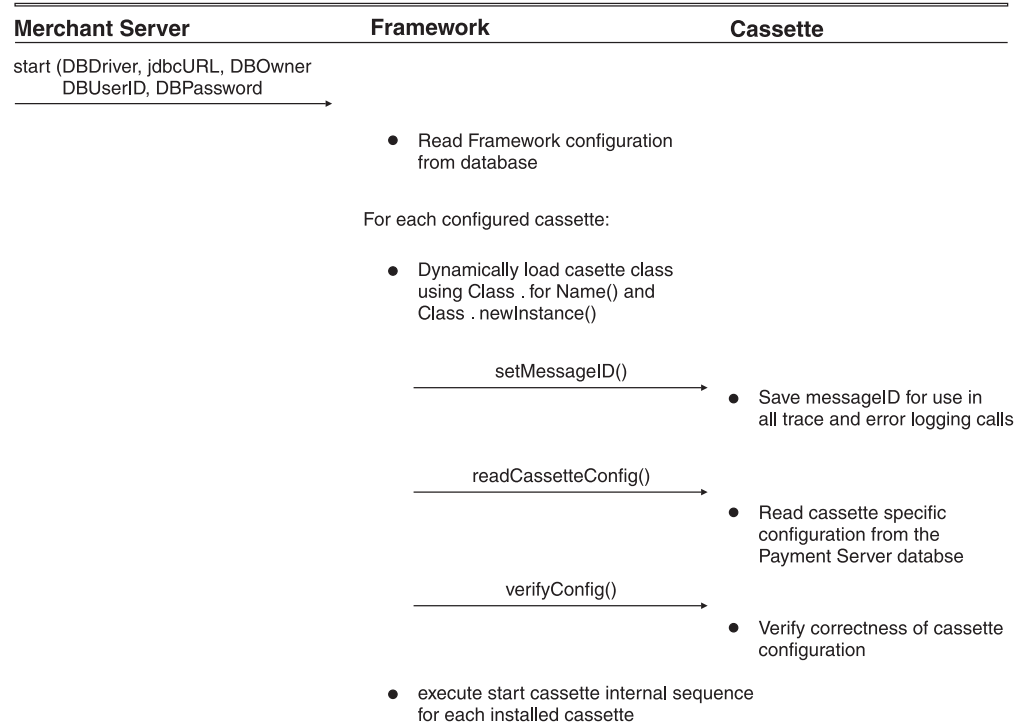
Sequence diagrams for administrative commands include:

- "System start sequence" on page 41
- "ModifyCassette API sequence" on page 42
- "StartCassette internal sequence" on page 43
- "StopCassette internal sequence" on page 44
- "CreateAccount API sequence" on page 45
- "ModifyAccount API sequence" on page 46
- "DeleteAccount API sequence" on page 47
- "StartAccount internal sequence" on page 48
- "StopAccount internal sequence" on page 48
- "DeleteAccount internal sequence" on page 49
- "CreatePaySystem API sequence" on page 50
- "ModifyPaySystem API sequence" on page 51
- "DeletePaySystem API sequence" on page 52
- "StartPaySystem internal sequence" on page 53
- "StopPaySystem internal sequence" on page 53
- "DeletePaySystem internal sequence" on page 54
- "CreateMerchant API sequence" on page 55
- "ModifyMerchant API sequence" on page 56
- "DeleteMerchant API sequence" on page 57
- "ModifyMerchantCassetteObject API sequence" on page 58
- "ModifySystemCassetteObject API sequence" on page 59
- "CassetteControl API sequence" on page 60

Administrative command sequence diagrams

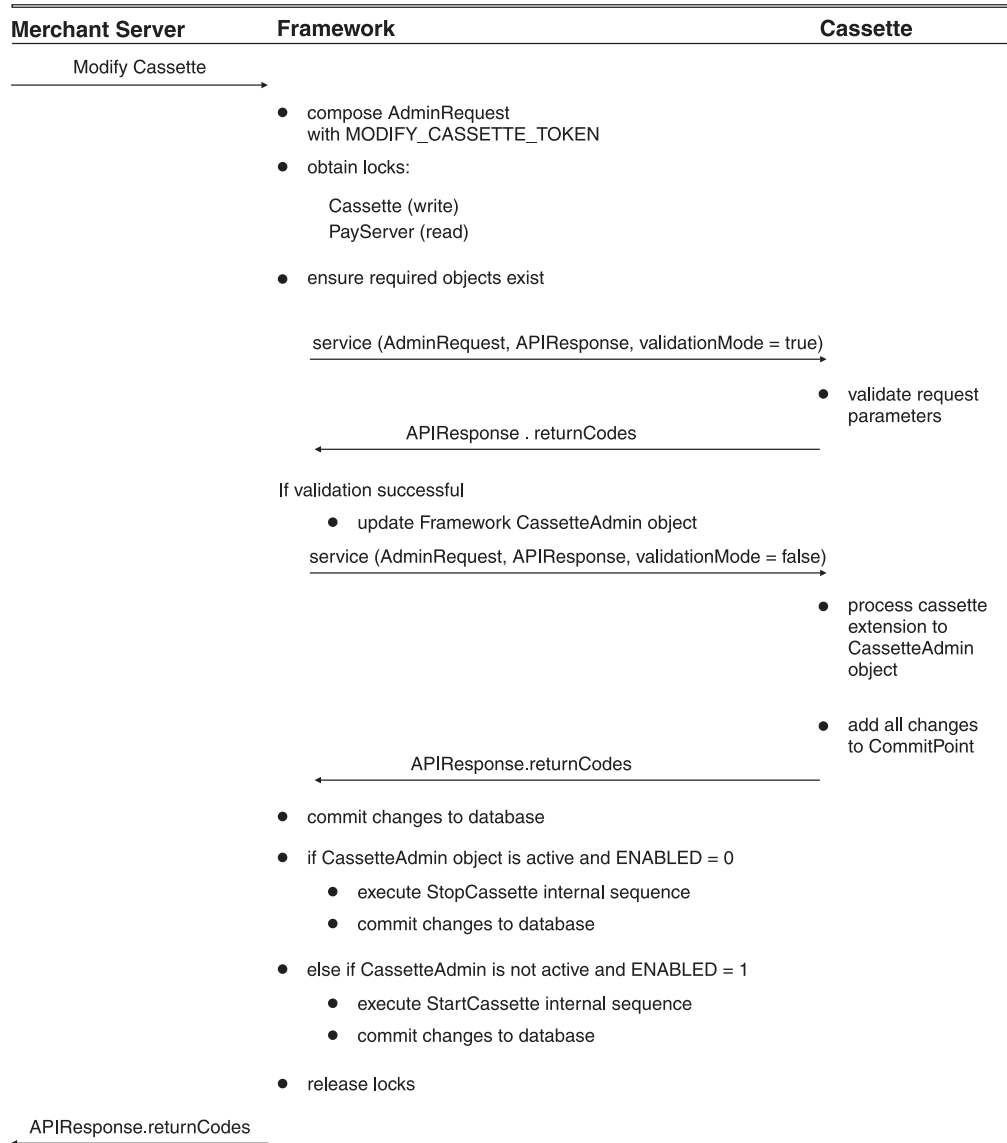
System start sequence

The Start sequence diagram shows the interactions between WebSphere Commerce Payments framework and cassette resulting from the Payment Servlet being started by the system administrator. The Start sequence only happens once. Note the call to “StartCassette internal sequence” on page 43.



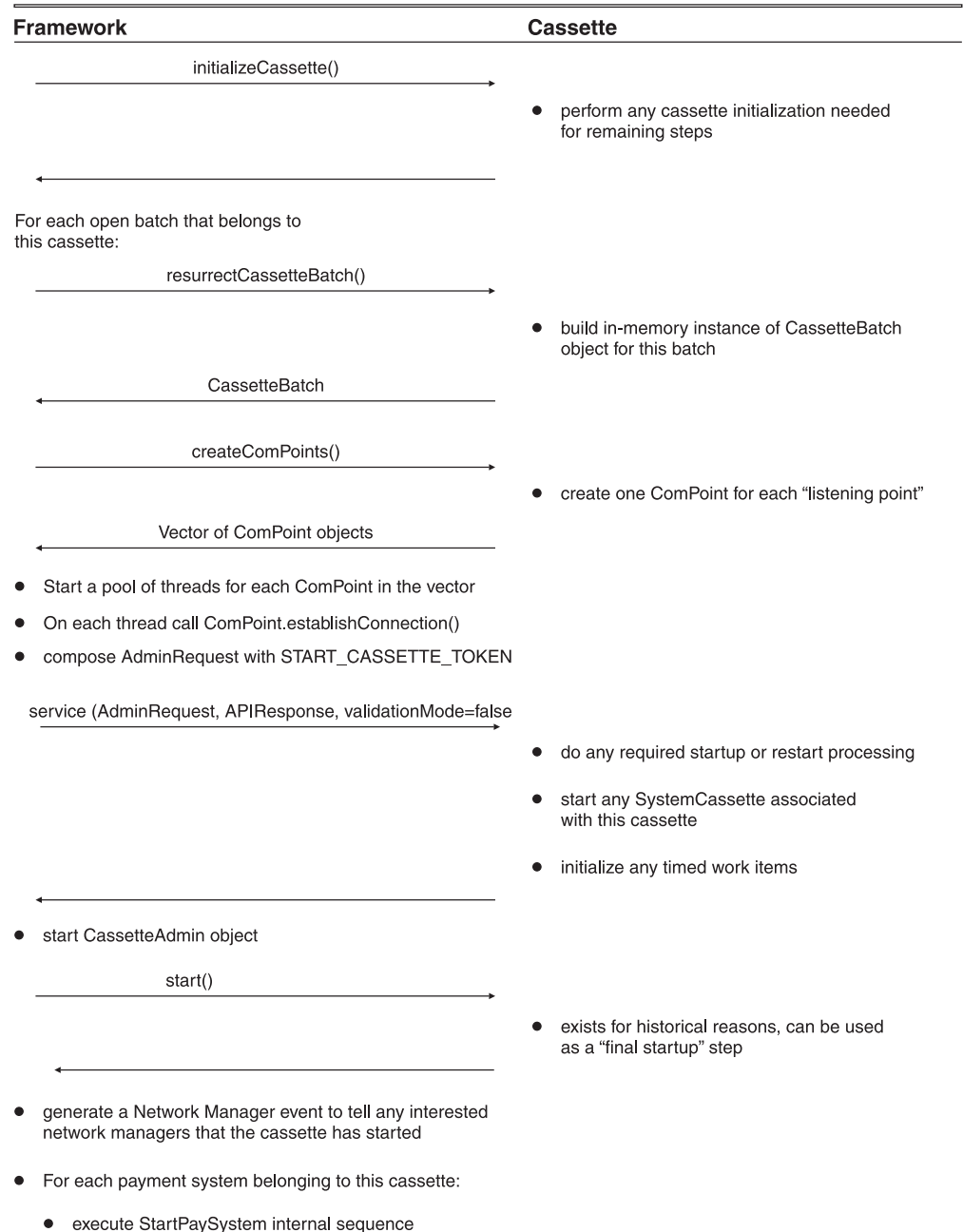
ModifyCassette API sequence

The ModifyCassette API sequence diagram shows the interactions between the WebSphere Commerce Payments framework and cassette when a WebSphere Commerce Payments administrator sends a ModifyCassette command. Note the calls to “StopCassette internal sequence” on page 44 and “StartCassette internal sequence” on page 43.



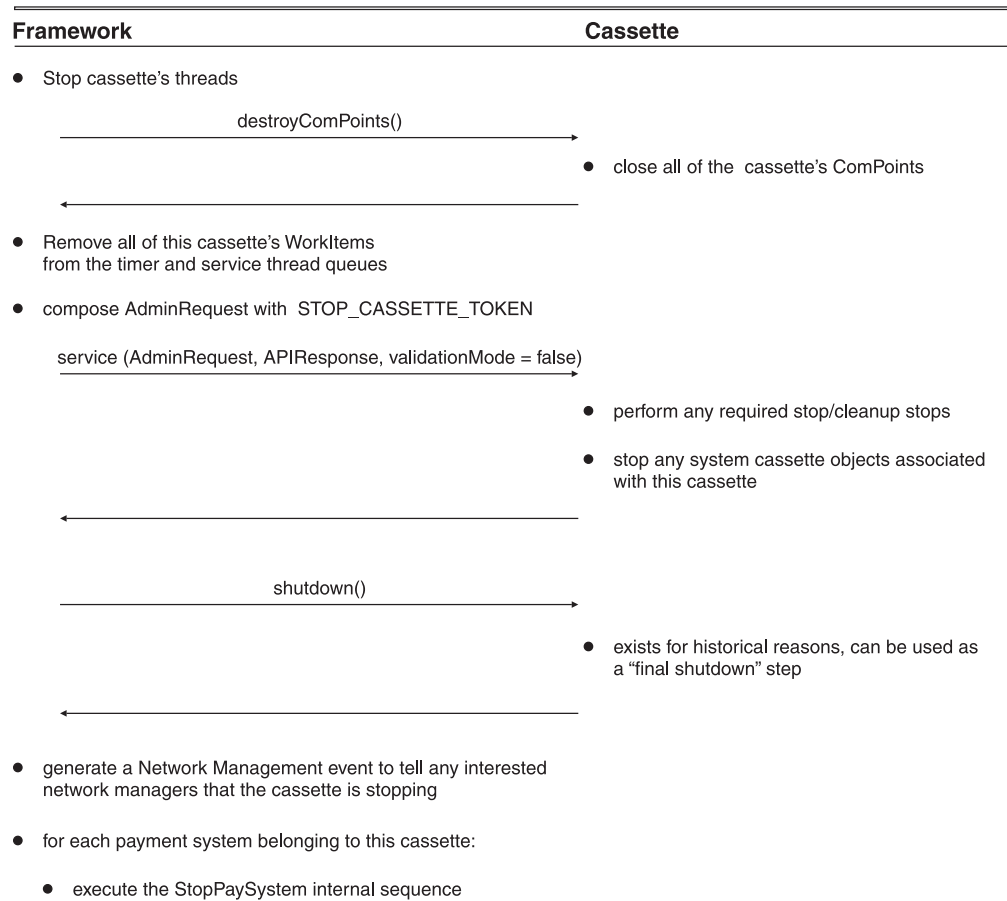
StartCassette internal sequence

The StartCassette internal sequence diagram shows the interaction that occurs when a cassette is started. This sequence is initiated when either the framework is initialized or when a cassette is enabled through a ModifyCassette command. When the framework is initialized, the ModifyCassette call is done.



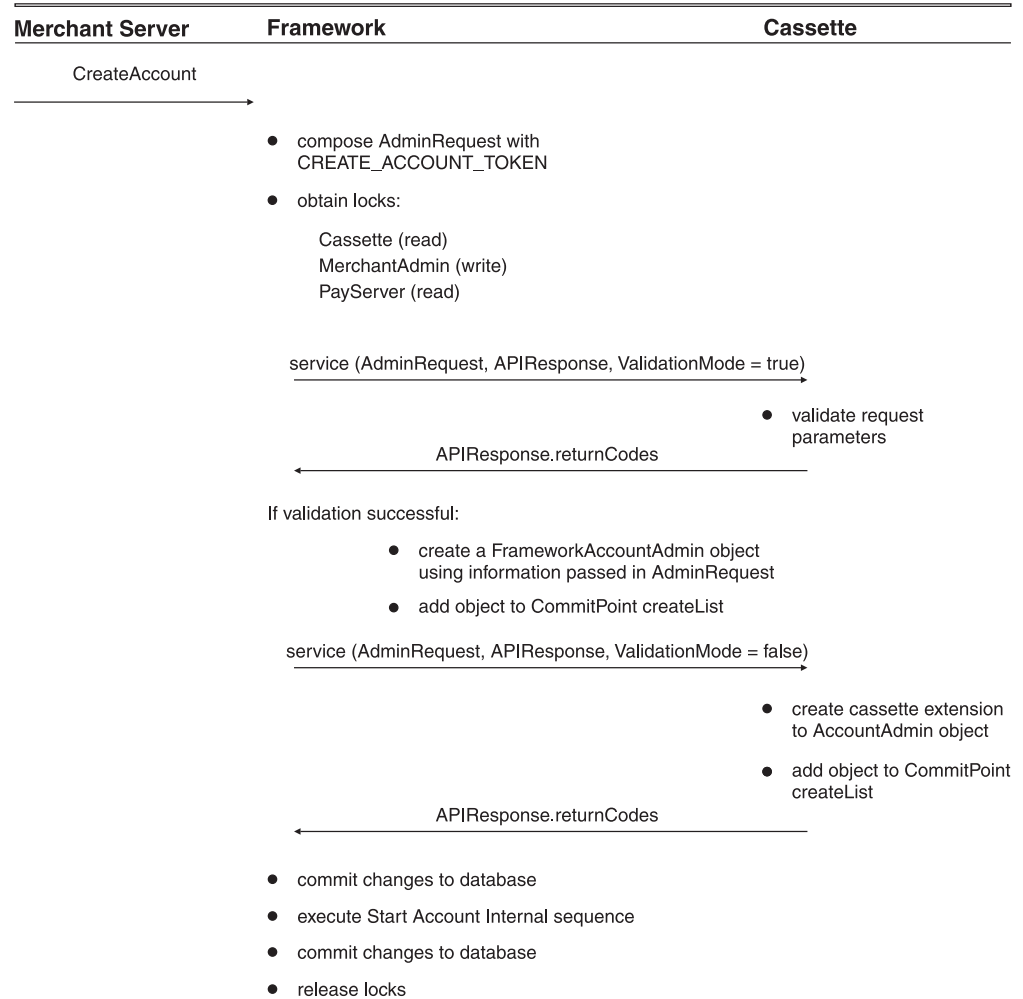
StopCassette internal sequence

The StopCassette sequence diagram shows the interactions between the WebSphere Commerce Payments framework and cassette resulting from WebSphere Commerce Payments being stopped by the system administrator.



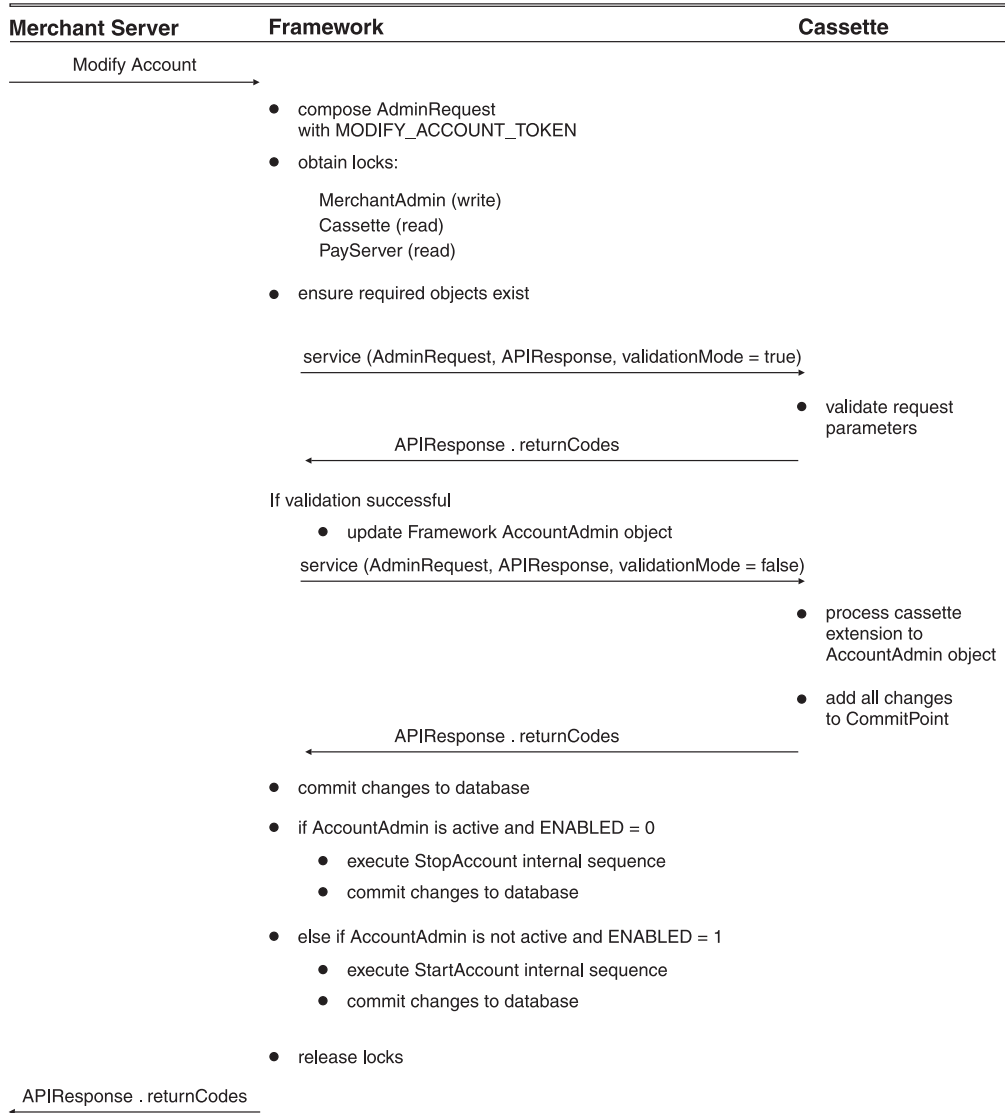
CreateAccount API sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when an account is created. Note the calls to “StopAccount internal sequence” on page 48 and “StartAccount internal sequence” on page 48.



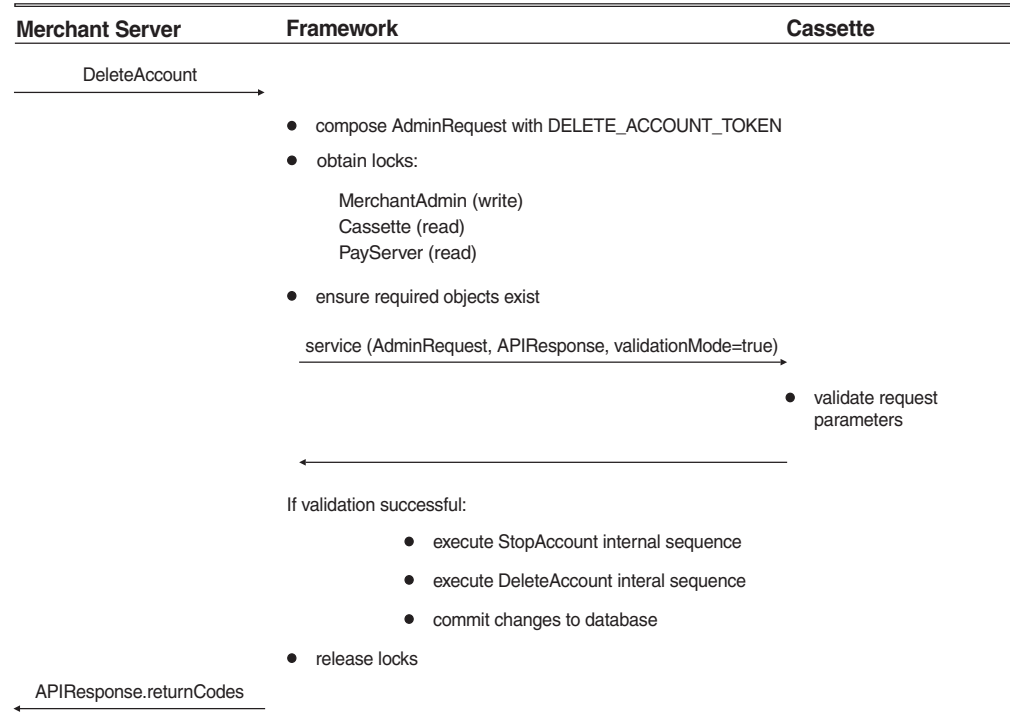
ModifyAccount API sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when an account is modified. Note the calls to “StopAccount internal sequence” on page 48 and “StartAccount internal sequence” on page 48.



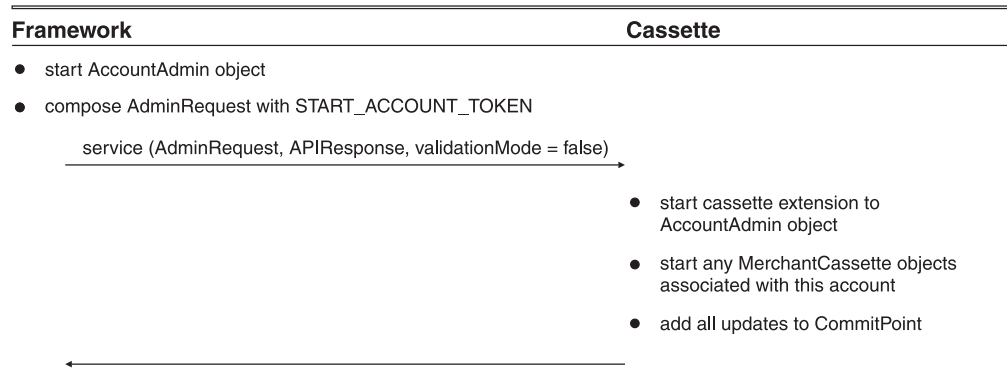
DeleteAccount API sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when an account is deleted. Note that this sequence includes calls to “StopAccount internal sequence” on page 48 and “DeleteAccount internal sequence” on page 49.



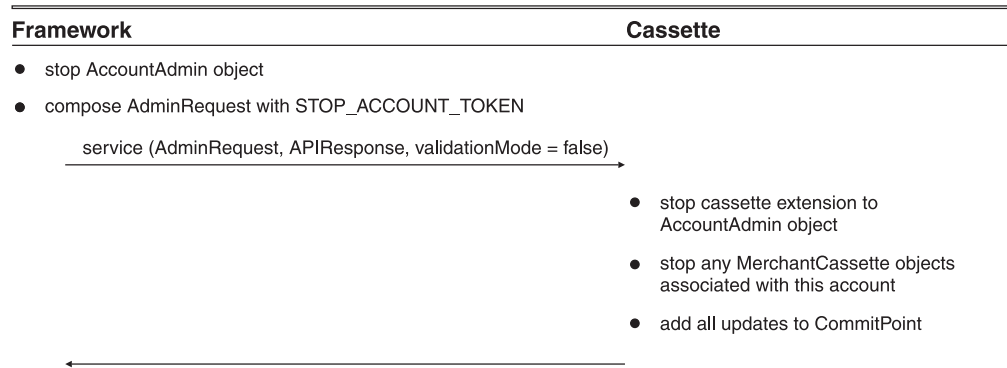
StartAccount internal sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when an account is started.



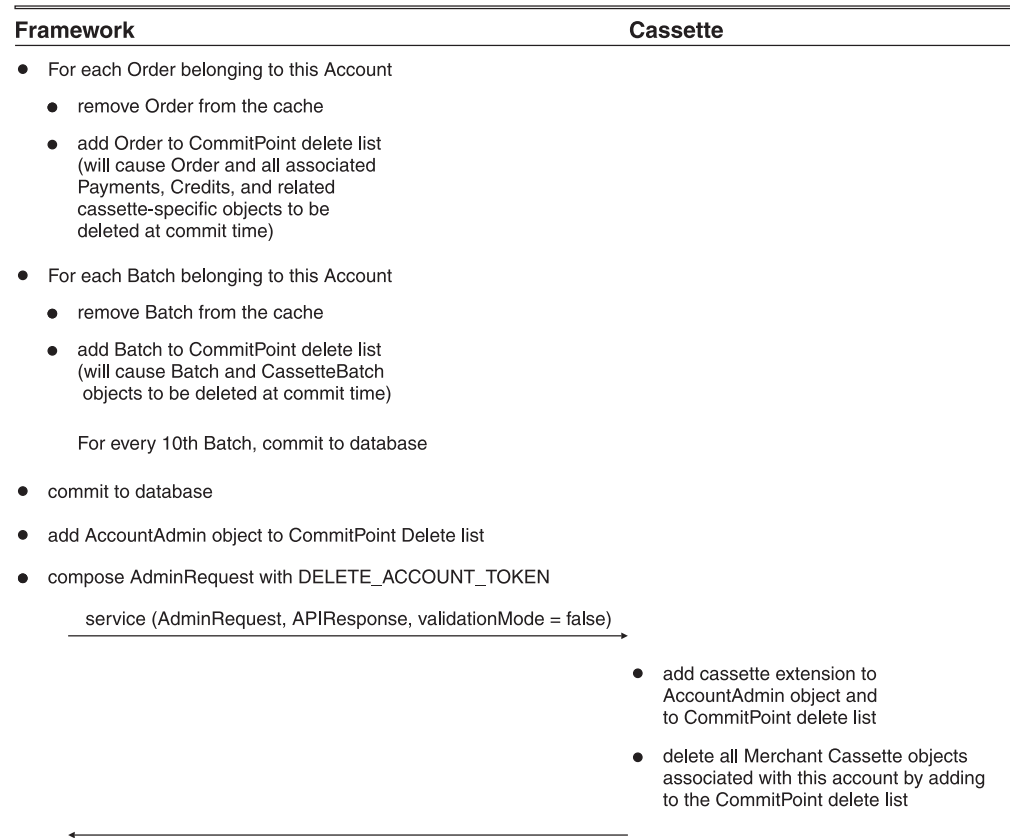
StopAccount internal sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when an account is stopped.



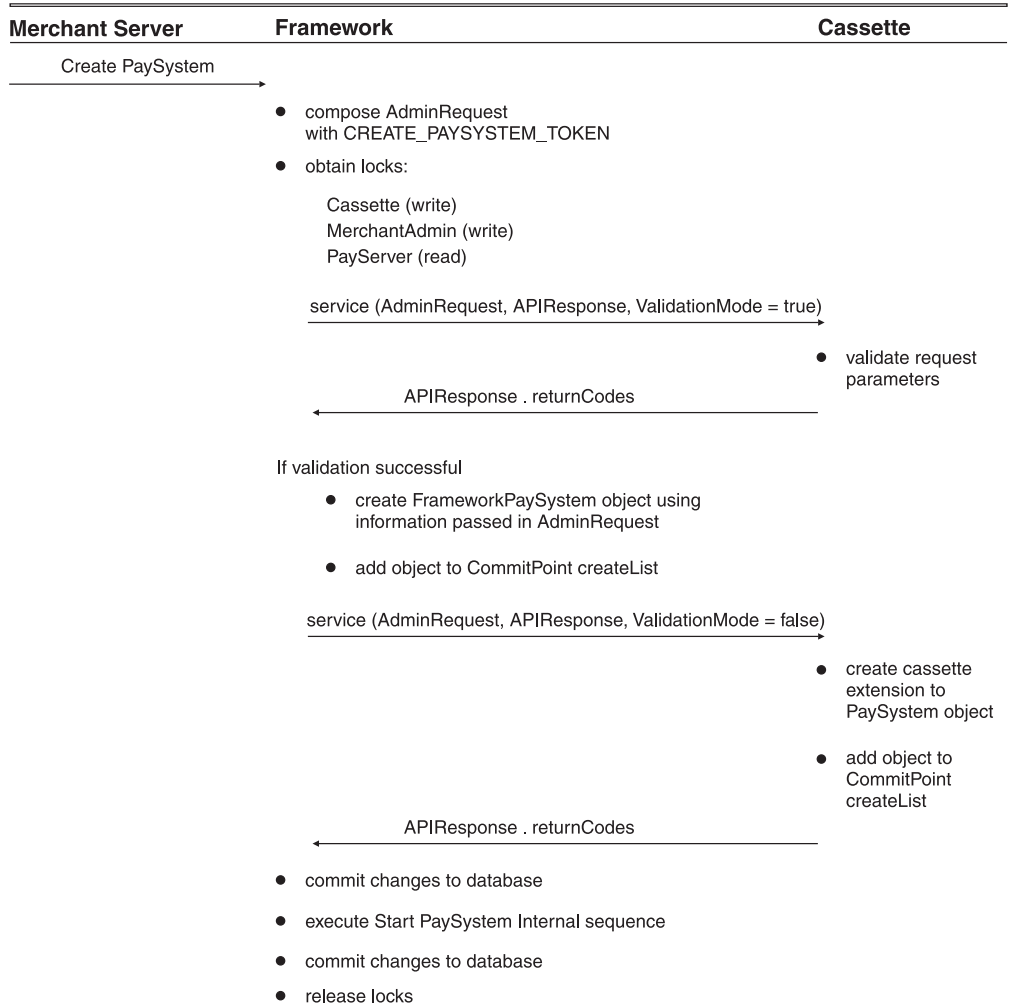
DeleteAccount internal sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when an Account is deleted.



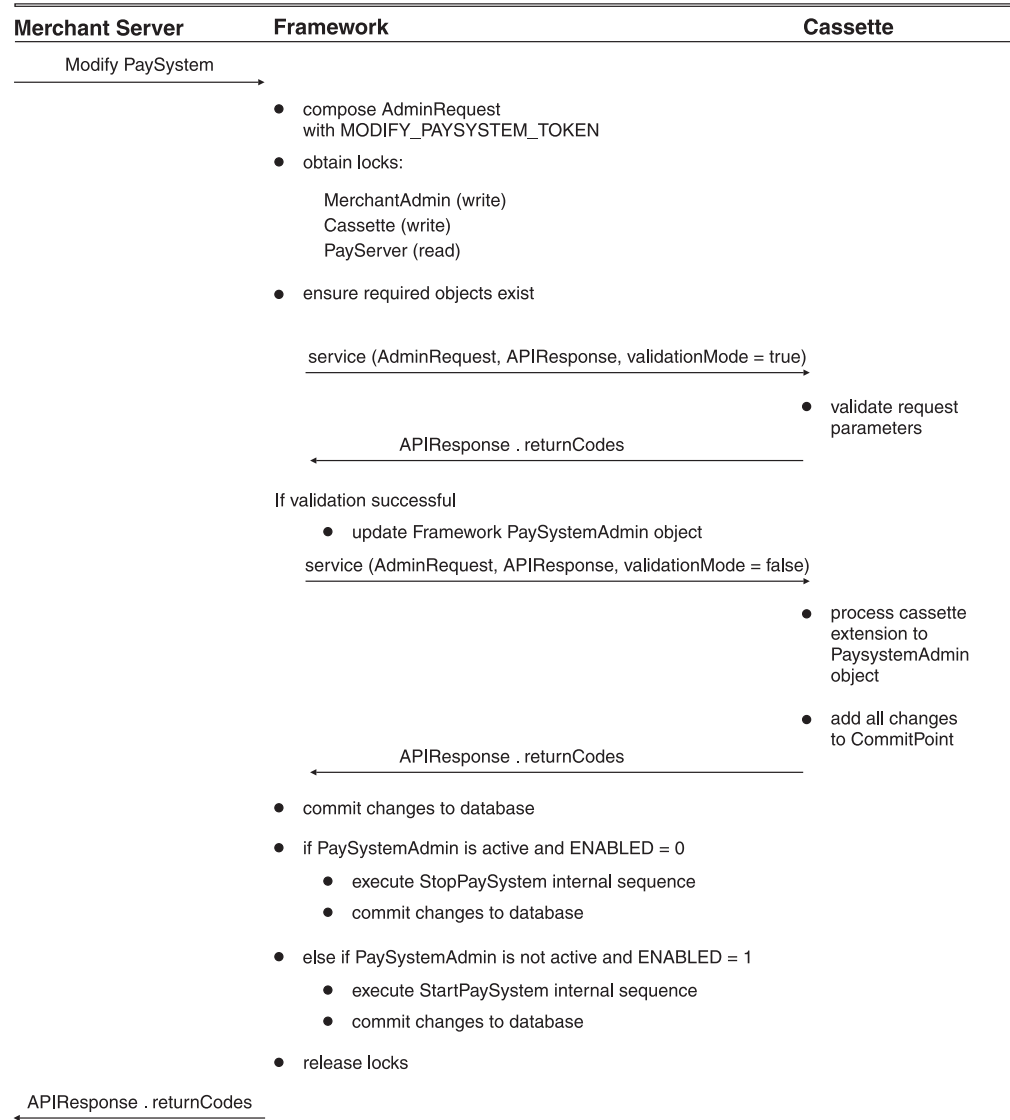
CreatePaySystem API sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when a PaySystem is created. Note the calls to “StopPaySystem internal sequence” on page 53 and “StartPaySystem internal sequence” on page 53.



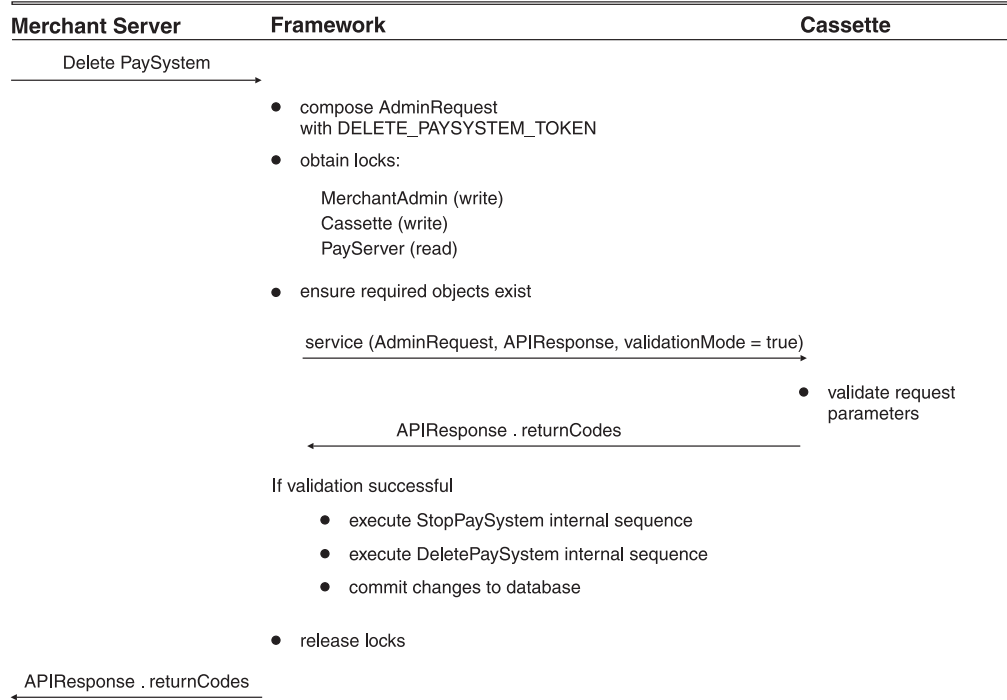
ModifyPaySystem API sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when a PaySystem is modified. Note the calls to “StopPaySystem internal sequence” on page 53 and “StartPaySystem internal sequence” on page 53.



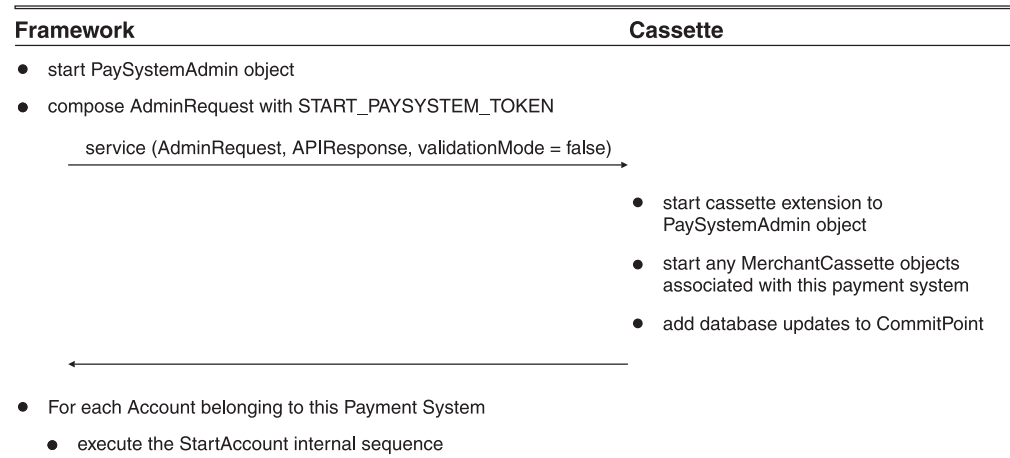
DeletePaySystem API sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when a PaySystem is deleted. Note the calls to “StopPaySystem internal sequence” on page 53 and “StartPaySystem internal sequence” on page 53.



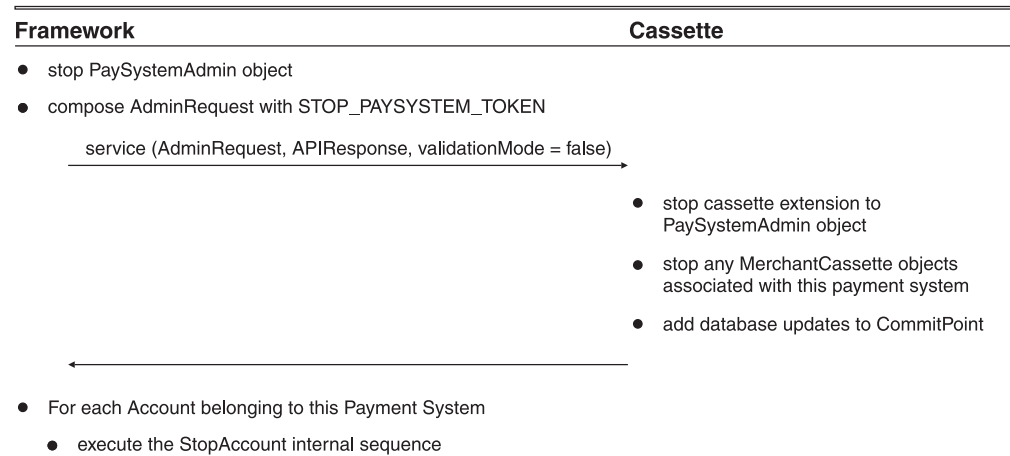
StartPaySystem internal sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when a PaySystem is started.



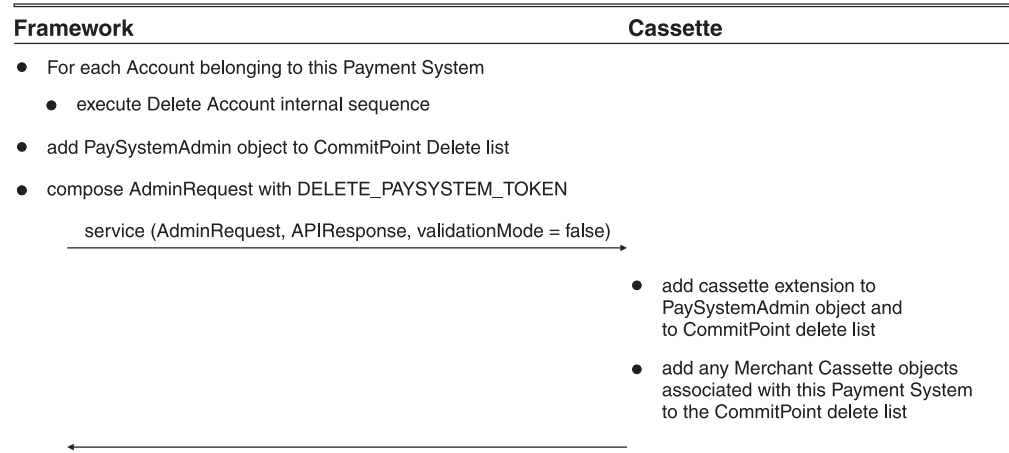
StopPaySystem internal sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when a PaySystem is stopped.



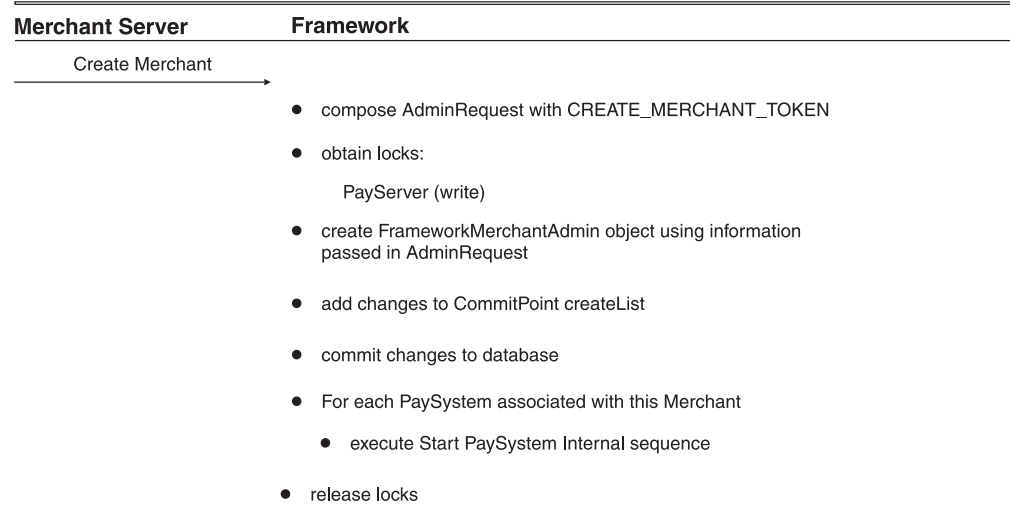
DeletePaySystem internal sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when a PaySystem is deleted.



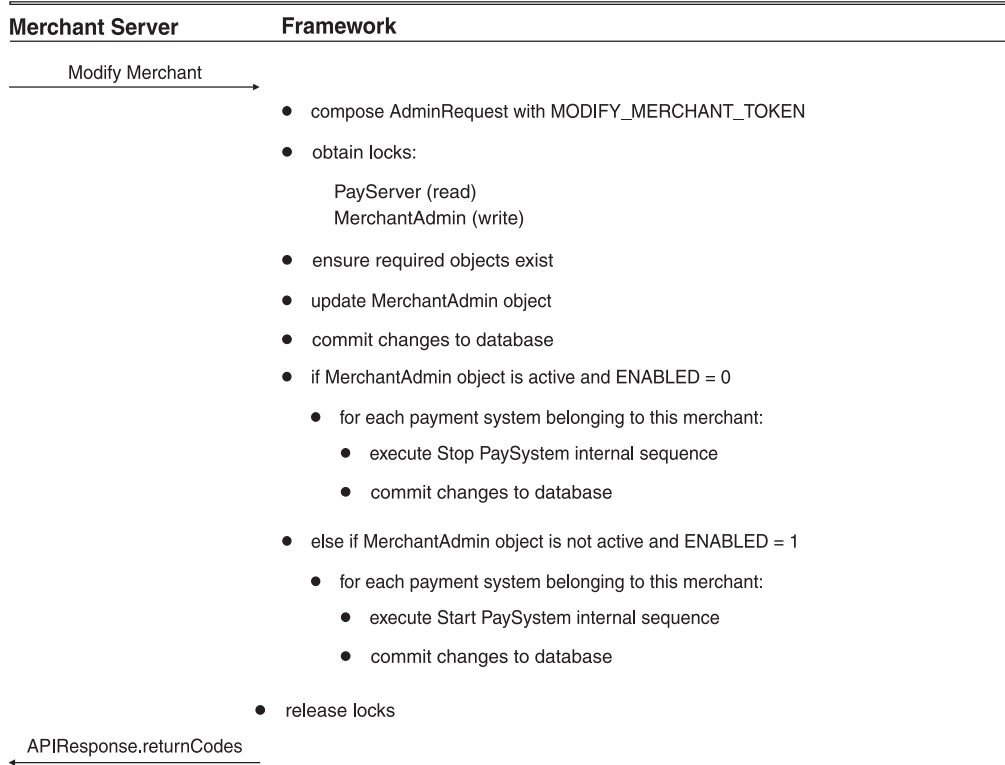
CreateMerchant API sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when a Merchant is created. Note the calls to “StopPaySystem internal sequence” on page 53 and “StartPaySystem internal sequence” on page 53.



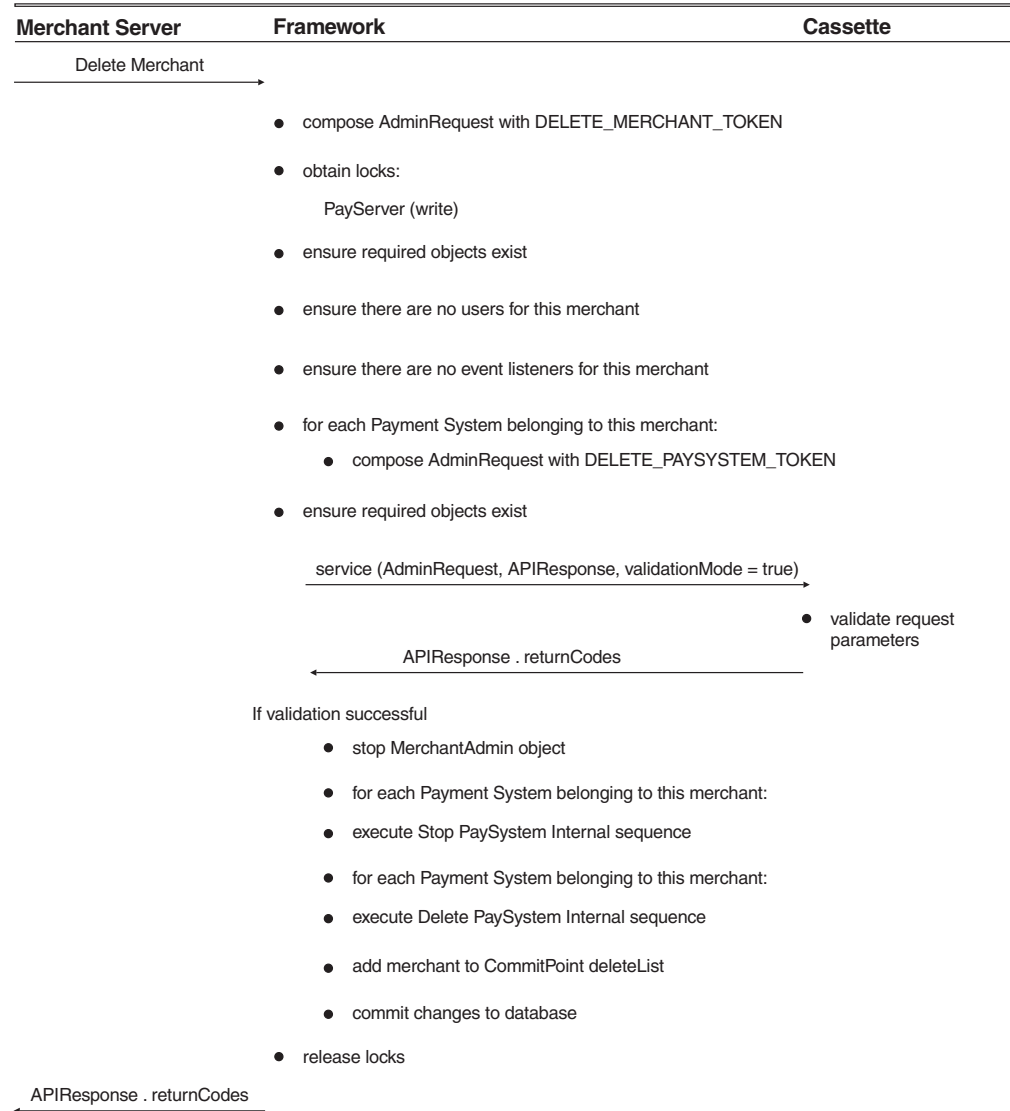
ModifyMerchant API sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when a Merchant is modified. Note the calls to “StopPaySystem internal sequence” on page 53 and “StartPaySystem internal sequence” on page 53.



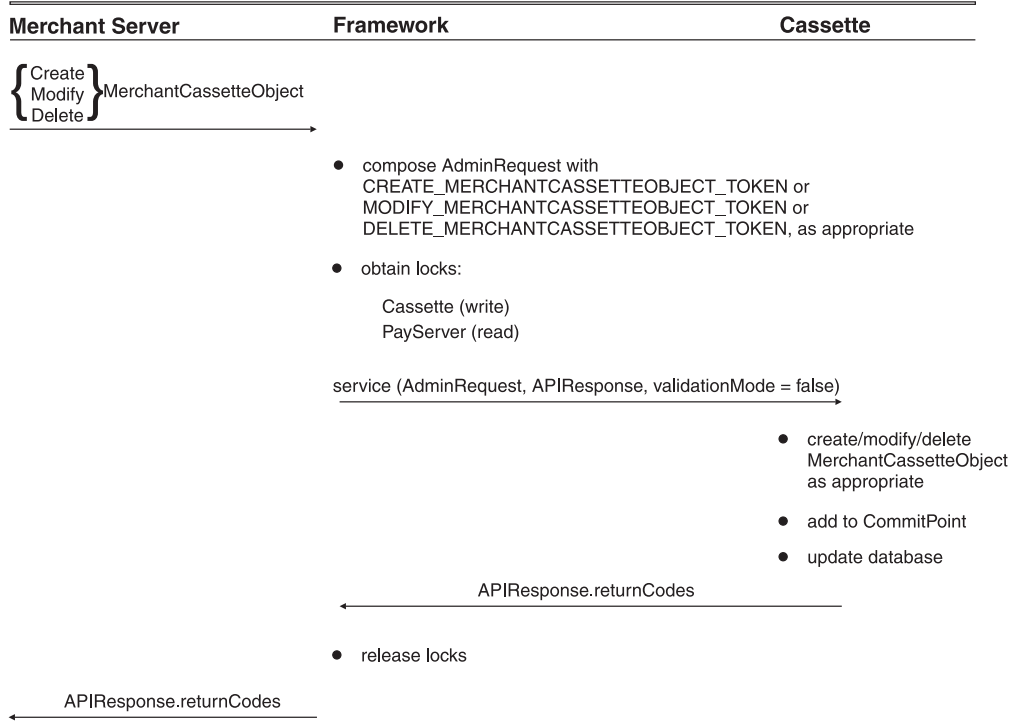
DeleteMerchant API sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when a Merchant is deleted. Note the calls to “StopPaySystem internal sequence” on page 53 and “StartPaySystem internal sequence” on page 53.



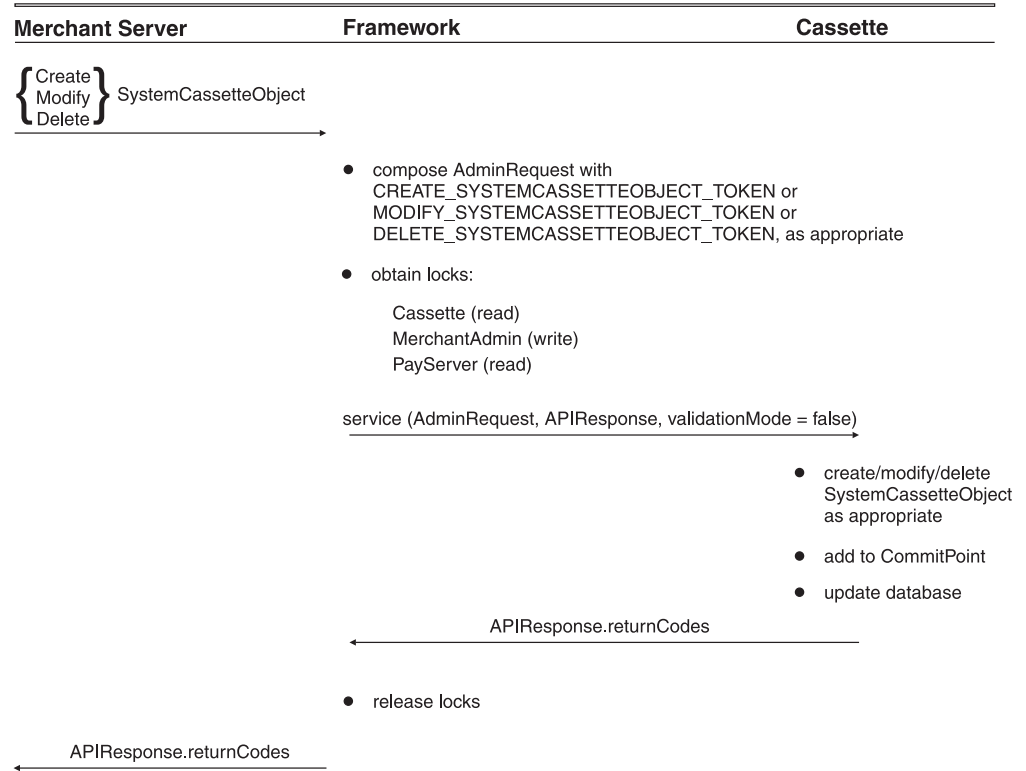
ModifyMerchantCassetteObject API sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when a MerchantCassetteObject is created, modified, or deleted. The interactions are the same for all three, except for the token being different.



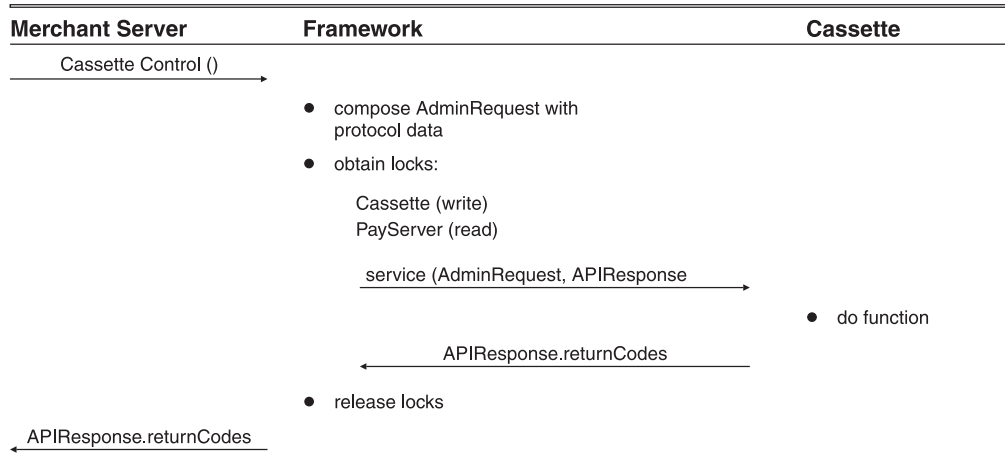
ModifySystemCassetteObject API sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when a SystemCassetteObject is created, modified, or deleted. The interactions are the same for all three, except for the token being different.



CassetteControl API sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and the cassette when a CassetteControl command is received from the merchant software.



Payment API commands

These commands request operations on financial objects. Not all commands need to cause action within your payment cassette; however every payment cassette **must** process either `ReceivePayment`, `AcceptPayment`, or both to initiate payment processing in the WebSphere Commerce Payments framework. All other payment commands are optional.

Note: This section describes the effect that each command has on the WebSphere Commerce Payments framework financial objects. When you map these commands onto your payment protocol, you will be able to provide a complete description of what each command means for your cassette.

As mentioned earlier in this chapter, if a command is not directly supported by your payment protocol but can be simulated, you should accept it. For example, you may be able to simulate an `ApproveReversal` by voiding an authorization and making a new authorization. If the unsupported command cannot be simulated, then reject it with `PRC_COMMAND_NOT_SUPPORTED`, `RC_NONE` instead of ignoring its existence. You should expect and tolerate this return code pair as a likely possibility and handle it accordingly.

ReceivePayment

`ReceivePayment` causes an order to be created. The framework will create the Order object and initialize it with the data received across the API. *The data received with this command is not sufficient to provide all payment instructions*, so it is assumed that additional data will be exchanged between consumer and cassette (using protocol messages) to fully define the payment. The semantics of this API require return of a complete HTTP response. The merchant server is expected to forward this response *as is* to the consumer. After creating and initializing the Order, the framework will ask the cassette to create a `CassetteOrder` object, then ask the cassette to service the `ReceivePayment` request. When successful processing of `ReceivePayment` is complete, the Order should be in `ORDER_REQUESTED` state.

The `ReceivePayment` command may be modified by the `AUTOAPPROVE` and `AUTODEPOSIT` flags. When `AUTOAPPROVE` equals 1, the cassette is expected to perform approve processing before completing the `ReceivePayment` command. When `AUTODEPOSIT` is true, the cassette is expected to perform deposit processing before completing the command. Unlike the `Approve` and `Deposit` commands, the framework will not create a Payment object when `AUTOAPPROVE` or `AUTODEPOSIT` flags are true on the `ReceivePayment` command. The cassette is expected to create a framework payment by calling the framework's `Supervisor.createPayment` method. This method will call the Cassette's `newCassettePayment` method.

AcceptPayment

`AcceptPayment` causes an Order to be created. The framework will create the Order object and initialize it with the data received across the API. *The data received with this command is sufficient to fully describe the payment method*. A cassette should respond to `AcceptPayment` rather than `ReceivePayment` when no further communication with the consumer is required to identify the complete payment instructions. After creating and initializing the Order, the framework will ask the cassette to create a `CassetteOrder` object and then ask the cassette to service the `AcceptPayment` request. When successful processing of `AcceptPayment` is complete, the Order should be in `ORDER_ORDERED` state, or the `ORDER_REFUNDABLE` state if the cassette supports independent credit.

The framework will not create a Payment object when the AUTOAPPROVE flag equals 1 or the AUTODEPOSIT flag is true on the AcceptPayment call. The cassette is expected to create a framework payment by calling the framework's Supervisor.createPayment method. This method will call the Cassette's newCassettePayment method. In addition, the cassette must commit the order to the database.

Approve

Approve causes a Payment object to be created. The framework will create and initialize the Payment and then ask the cassette to create a cassette-specific payment (type is derived from CassetteTransaction). The cassette will then be asked to service the Approve request. When an Approve command is processed successfully, the cassette must commit the framework Payment object to the database with a state of PAYMENT_APPROVED, PAYMENT_DEPOSITED, or PAYMENT_CLOSED, depending upon the payment protocol and cassette implementation.

ApproveReversal

ApproveReversal either voids an approval or replaces the amount of an approval with the amount on the ApproveReversal command. If this amount is zero, the approval is voided (that is, a full reversal). Otherwise, the command operates as an approve adjustment (a partial reversal). The framework will look up the Order and Payment and then ask the cassette to service the ApproveReversal request. Cassette writers must decide whether or not their cassettes should support the ApproveReversal command. If they do support the command, they must also decide whether to support full reversals, partial reversals, or both. All of these decisions should be based upon the needs of their payment protocol and the needs of the merchant software that will eventually use the cassette:

- If approve reversals are not supported by the payment protocol, then the cassette should determine if it wants to support the command as a local operation (i.e., local in the sense that there is no communication to a payment processor). Depending on the payment protocol, this may be necessary since it allows the merchant server software to indicate that the payment should be in PAYMENT_VOID state. Not supporting ApproveReversal will leave the payment in PAYMENT_APPROVED state, and leave the order in a state that does not allow it to be cancelled via the CancelOrder command. If the determination is made that the cassette will not support approve reversals, local or otherwise, then the cassette should throw an ETillAbortOperation exception with primary and secondary return codes set to PRC_COMMAND_NOT_SUPPORTED and RC_NONE, respectively.
- If the reversal is supported, the cassette must take whatever action is necessary to modify or void the associated approval, and it must commit the Payment object with a state of PAYMENT_VOID (if voided) or PAYMENT_APPROVED (if partially reversed). If the reversal cannot be completed successfully due to some error, the state of the Payment object should revert back to PAYMENT_APPROVED.
- If the reversal is declined by the bank, the state of the Payment object should go to PAYMENT_DECLINED.

Deposit

Deposit arranges for the deposit of funds against a previously approved Payment. The framework will look up the Order and Payment and then ask the cassette to service the Deposit request. Cassette writers must decide whether or not their cassettes should support the Deposit command. This decision should be based on the needs of their payment protocol:

- If `Deposit` is not supported then the cassette should throw an `ETillAbortOperation` exception with primary and secondary return codes set to `PRC_COMMAND_NOT_SUPPORTED` and `RC_NONE`, respectively.
- If `Deposit` is supported, then the cassette must take whatever action is necessary to process the deposit request, including any necessary associations with batches if the cassette supports batch processing. Upon successful completion of this request, the cassette must commit the framework `Payment` object with a state of `Payment_Deposit` or `PAYMENT_CLOSED`, depending upon the payment protocol and the cassette implementation. If the `Deposit` cannot be completed successfully due to some error, the state of the `Payment` object should revert back to `PAYMENT_APPROVED`.

DepositReversal

`DepositReversal` voids a deposit that was issued for the associated `Payment` object in `PAYMENT_DEPOSITED` state. When a `DepositReversal` command is received, the framework will look up the `Order` and `Payment` and then ask the cassette to service the `DepositReversal` request. The cassette makes the final decision concerning whether such reversals are supported at all:

- If deposit reversals are not supported then the cassette should throw an `ETillAbortOperation` exception with primary and secondary return codes set to `PRC_COMMAND_NOT_SUPPORTED` and `RC_NONE`, respectively.
- If the reversal is supported, then the cassette must take whatever action is necessary to void the associated deposit, and it must commit the `Payment` object with a state of `PAYMENT_VOID`. If the reversal cannot be completed successfully due to some error, the state of the `Payment` object should revert back to `PAYMENT_DEPOSITED`.
- Only full reversals are supported. Partial reversals are not allowed.

Refund

`Refund` creates a `Credit` object for an `Order` that is in `ORDER_REFUNDABLE` state. The framework will create and initialize the `Credit` and then ask the cassette to create a cassette-specific credit (type is derived from `CassetteTransaction`). The cassette will then be asked to service the `Refund` request. Cassette writers must decide whether or not their cassettes should support the `Refund` command. This decision should be based on the needs of their payment protocol:

- If `Refund` is not supported then the cassette should throw an `ETillAbortOperation` exception with primary and secondary return codes set to `PRC_COMMAND_NOT_SUPPORTED` and `RC_NONE`, respectively.
- If `Refund` is supported, then the cassette must take whatever action is necessary to process the refund request, including any necessary associations with batches if the cassette supports batch processing. Upon successful completion of this request, the cassette must commit the framework `Credit` object with a state of `CREDIT_REFUNDED` or `CREDIT_CLOSED`, depending upon the payment protocol and the cassette implementation. If the `Refund` cannot be completed successfully due to some error, the state of the `Payment` object should be set to `CREDIT_DECLINED` (for financial rejection). Use `CREDIT_VOID` for other errors.

RefundReversal

`RefundReversal` voids a refund that was issued for the associated `Credit` object (in `CREDIT_REFUNDED` state). When a `RefundReversal` command is received, the framework will look up the `Order` and `Credit` and then ask the cassette to service the `RefundReversal` request. The cassette makes the final decision concerning whether such reversals are supported at all:

- If refund reversals are not supported then the cassette should throw an `ETillAbortOperation` exception with primary and secondary return codes set to `PRC_COMMAND_NOT_SUPPORTED` and `RC_NONE`, respectively.
- If the reversal is supported, then the cassette must take whatever action is necessary to void the associated refund and it must commit the `Credit` object with a state of either `CREDIT_PENDING` (if the request was forwarded on to some third party and the result has not yet been received) or `CREDIT_VOID` (if the reversal is complete). If the reversal cannot be completed successfully due to some error, the state of the `Credit` object should revert back to `CREDIT_REFUNDED`.
- Refund reversals are only allowed if the `Order` object is in `ORDER_REFUNDABLE` state and the `Credit` object is in `CREDIT_REFUNDED` state.
- Only full reversals are supported. Partial reversals are not allowed.

CloseOrder

`CloseOrder` causes an `Order` that owns at least one closed `Payment` or one closed `Credit` to become unavailable for any further financial transactions. If the `DELETEORDER` option was specified, then the `Order` object and all of its related ancillary objects are purged from the database. When a `CloseOrder` command is received, the framework will look up the `Order` and then ask the cassette to service the `CloseOrder` request. Every cassette should support `CloseOrder`.

- If `DELETEORDER` is not specified, then the only thing required of the cassette is that the `Order` object be committed with a state of `ORDER_CLOSED`.
- If `DELETEORDER` is specified, then the cassette must also delete the framework `Order` object by calling the `com.ibm.etill.framework.supervisor.Supervisor.removeOrder` method. This method will cause several cassette methods to be called in order to delete all of the cassette objects associated with the `Order` and its `Payment` and `Credit` objects.

CancelOrder

`CancelOrder` makes an `Order` that has either (1) no `Payments` or `Credits`, or (2) all associated `Payments` or `Credits` in their respective `RESET`, `VOID` or `DECLINED` state unavailable for any further financial transactions. If the `DELETEORDER` option was specified, then the `Order` object and all of its related ancillary objects are purged from the database. When a `CancelOrder` command is received, the framework will look up the `Order` and then ask the cassette to service the `CancelOrder` request. Every cassette that maintains the `PAYMENT_DEPOSITED` state for `Payments` should support `CancelOrder`.

- If `DELETEORDER` is not specified, then the only thing required of the cassette is that the `Order` object be committed with a state of `ORDER_CANCELED`.
- If `DELETEORDER` is specified, then the cassette must also delete the framework `Order` object by calling the `com.ibm.etill.framework.supervisor.Supervisor.removeOrder` method. This method will cause several cassette methods to be called in order to delete all of the cassette objects associated with the `Order` and its `Payment` and `Credit` objects.
- `CancelOrders` are allowed only if the object is in `ORDER_ORDERED`, `ORDER_REJECTED`, `ORDER_REFUNDABLE`, or `ORDER_CANCELED` states.

BatchOpen

`BatchOpen` causes a `Batch` to be created. The framework will create and initialize a `Batch` and then ask the cassette to create a cassette-specific `Batch` object. The cassette is then asked to service the `BatchOpen` request. After successful processing

of the BatchOpen request, a Batch is in Open state. Cassette writers must decide whether or not their cassettes should support the BatchOpen command. This decision should be based on the needs of their payment protocol:

- If BatchOpen is not supported, the cassette should throw an `ETillAbortOperation` exception with primary and secondary return codes set to `PRC_COMMAND_NOT_SUPPORTED` and `RC_NONE`, respectively.
- If BatchOpen is supported, then the cassette must take whatever action is necessary to process the open request, including any necessary communications with an external financial processor. Upon successful completion of this request, the cassette must commit the framework Batch object with a state of `BATCH_OPEN`. If the request cannot be completed successfully due to some error, the Batch object should be deleted from the database.

BatchClose

`BatchClose` causes an open Batch to be closed. It is not required that an explicit `BatchOpen` command be issued to be able to issue `BatchClose`.

The framework will locate the Batch and then ask the cassette to process the request. The cassette is then asked to service the `BatchClose` request. After successful processing of the `BatchClose` request, a Batch is in `BATCH_CLOSED` state. Again, cassette writers must decide whether or not their cassettes should support the `BatchClose` command. This decision should be based on the needs of their payment protocol:

- If `BatchClose` is not supported, the cassette should throw an `ETillAbortOperation` exception with primary and secondary return codes set to `PRC_COMMAND_NOT_SUPPORTED` and `RC_NONE`, respectively.
- If `BatchClose` is supported, then the cassette must take whatever action is necessary to close the Batch, including any necessary communications with an external financial processor. Upon successful completion of this request, the cassette must:
 - Commit the framework Batch object with a state of `BATCH_CLOSED`, as well as with the appropriate batch status value.
 - Close associated Payments and Credits.
 - If the request cannot be completed successfully due to some error, the Batch object should revert back to `BATCH_OPEN`.

DeleteBatch

`DeleteBatch` removes a closed Batch and all of its associated ancillary objects from the WebSphere Commerce Payments database. When `DeleteBatch` is received, the framework will look up the Batch and then ask the cassette to service the request. If the cassette supports batch operations, then it should support this request:

- If `DeleteBatch` is not supported, the cassette should throw an `ETillAbortOperation` exception with primary and secondary return codes set to `PRC_COMMAND_NOT_SUPPORTED` and `RC_NONE`, respectively.
- If `DeleteBatch` is supported, the cassette must delete the Batch object by calling the `com.ibm.etill.framework.supervisor.Supervisor.removeBatch` method. This method will cause one or more cassette methods to be called in order to delete all of the cassette objects associated with the Batch.

BatchPurge

`BatchPurge` removes all Payments and Credits from a batch. The Batch object is returned to Open state. The Payment objects are returned to Approved state and the Credit objects are returned to Void state. Deposits and Refunds can then be issued to reconstruct the batch. When `BatchPurge` is received, the framework will

look up the Batch and then ask the cassette to service the request. If the cassette supports batch operations, then it should support this request

- If BatchPurge is not supported, the cassette should throw an ETillAbortOperation exception with primary and secondary return codes set to PRC_COMMAND_NOT_SUPPORTED and RC_NONE, respectively.
- If BatchPurge is supported, the cassette must remove all Payments and Credits from the batch.

Payment protocol mapping

How your payment protocol maps to the WebSphere Commerce Payments payment API will depend very much on the type of payment protocol:

- cash
- check
- credit/debit
- micro-payment

This table shows the possible mappings to the WebSphere Commerce Payments API for each category of payment protocol.

Table 3. Payment API commands. An asterisk identifies commands not currently implemented by WebSphere Commerce Payments.

API command	Credit/Debit	Check	Cash	Micro Payment
AcceptPayment	initiate an order with all necessary consumer data. Used when no response to consumer is required.	process an e-check received using e-mail, or other source	not used	process a payment received as part of resolving a hypertext link
Approve	authorize	not used, except if third party guarantee is implemented	not used	obtain guarantee from consumer's billing server
ApproveReversal	authorize reversal	not used, except if third party guarantee is implemented	not used	not used
*Authenticate	possibly used to authenticate purchasing cards	not used	exchange authentication data with consumer	not used
BatchOpen	open a batch	open a batch ("deposit")	open a batch	open a batch
BatchClose	close a batch	close a batch and send a "deposit" to the bank	close a batch and transmit all received funds to the bank	close a batch and transmit all received payments to the bank
Control	not used	control local checkbook functions	control local cash functions	not used
Deposit	capture	endorse a check and add it to a deposit batch	not used	assign a payment to a batch
CloseOrder	close the order in the local database	close the order in the local database	close the order in the local database	close the order in the local database
CancelOrder	cancel the order in the local database	cancel the order in the local database	cancel the order in the local database	cancel the order in the local database
DeleteBatch	delete the batch from the local database	delete the batch from the local database	delete the batch from the local database	delete the batch from the local database

Table 3. Payment API commands (continued). An asterisk identifies commands *not* currently implemented by WebSphere Commerce Payments.

API command	Credit/Debit	Check	Cash	Micro Payment
DepositReversal	capture reversal	remove a check from a deposit batch	remove a payment from a batch	remove a payment from a batch
*Pay	initiate an "independent credit"	generate an e-check	make a payment to someone	generate a credit to someone
*PayReversal	reverse an "independent credit"	cancel an e-check	not used	reverse a credit
Refund	credit	not used	make a payment as a refund for a payment previously received	generate a credit against a payment previously received
RefundReversal	credit reversal	not used	not used	reverse a refund
ReceivePayment	initiate an order when consumer data still needs to be acquired. Used when an initiation message must be returned to the consumer.	not used	start a payment and generate a wallet kickoff message	not used
*Withdraw	not used	not used	obtain e-cash from a bank (that is, to provide funds for payments)	not used

Payment command sequence diagrams

WebSphere Commerce Payments responds to events from the outside world. Scenarios describe the processing that occurs as a result of the receipt of a particular event. We use sequence diagrams to visually describe the sequence of interactions between the major participants for a particular scenario. The sequence diagrams here describe the success scenarios for each of the events that WebSphere Commerce Payments handles.

For all of the Payment commands, the main interface between the framework and the cassette is the Cassette object's service method.

The details in the sequence diagrams are not intended to be absolutely precise. For instance, method calls will use the real name of the Java method, but will not precisely define the parameters to that method. The diagrams are intended to give a logical idea of the responsibilities of the framework and cassette for each scenario.

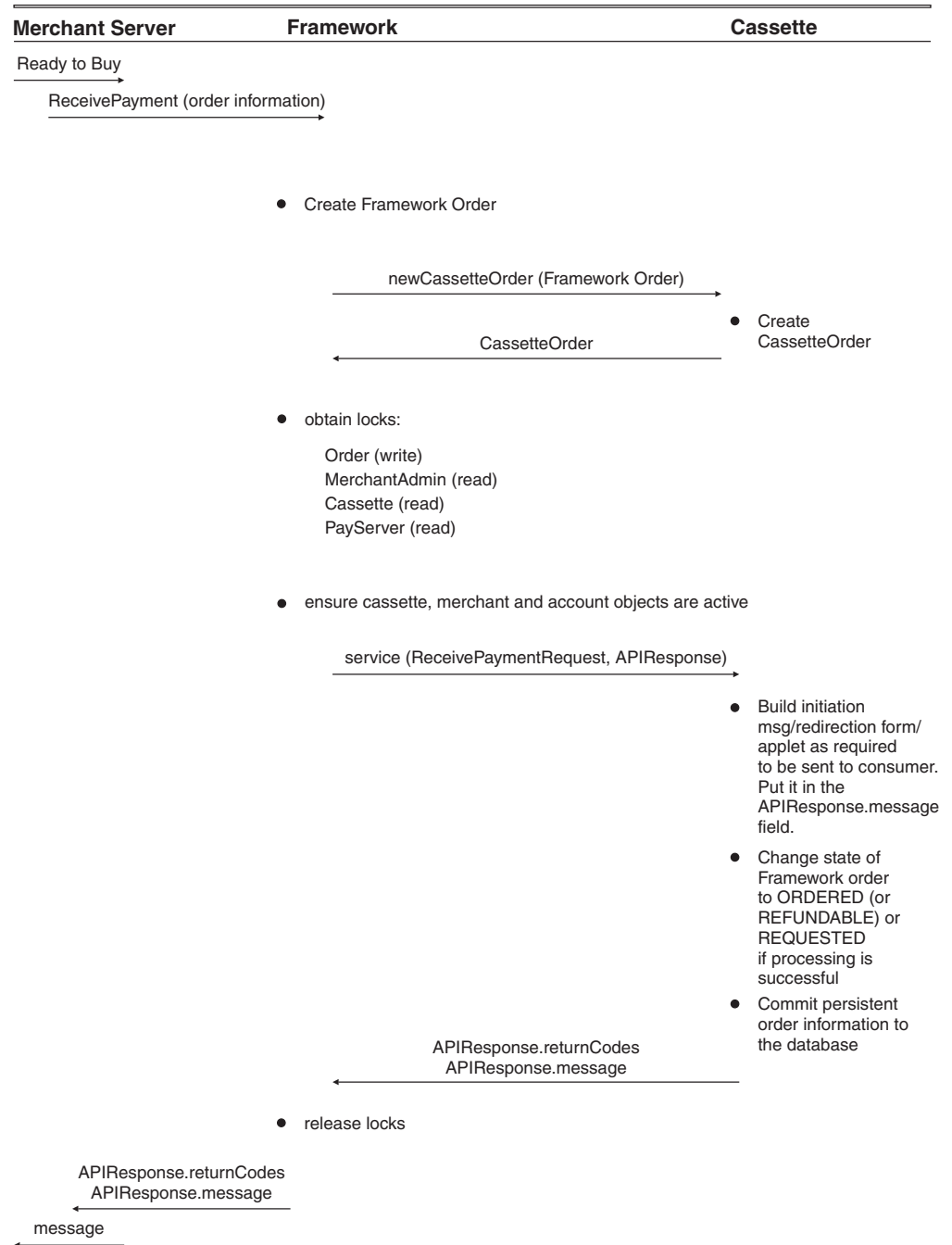
- "ReceivePayment API sequence" on page 69
- "Deposit API sequence" on page 74
- "DepositReversal API sequence" on page 75
- "Refund API sequence" on page 76
- "RefundReversal API sequence" on page 77
- "BatchOpen API sequence" on page 78
- "DeleteBatch API sequence" on page 80
- "BatchPurge API sequence" on page 81
- "CloseOrder API control sequence" on page 82

- “CancelOrder API sequence” on page 83
- “Protocol message API sequence” on page 84
- Key internal sequences:
 - “CreateBatch internal sequence” on page 85
 - “RetrieveBatch internal sequence” on page 86
 - “RetrieveOrder internal sequence” on page 87
 - “Service queue internal sequence” on page 88
 - “Timer queue internal sequence” on page 89

ReceivePayment API sequence

The ReceivePayment sequence diagram shows the interactions between the WebSphere Commerce Payments framework and cassette resulting from a ReceivePayment command from the merchant shopping software to WebSphere Commerce Payments.

If the cassette supports independent credit, the cassette changes the state of the framework order to REFUNDABLE (rather than ORDERED) if processing is successful.

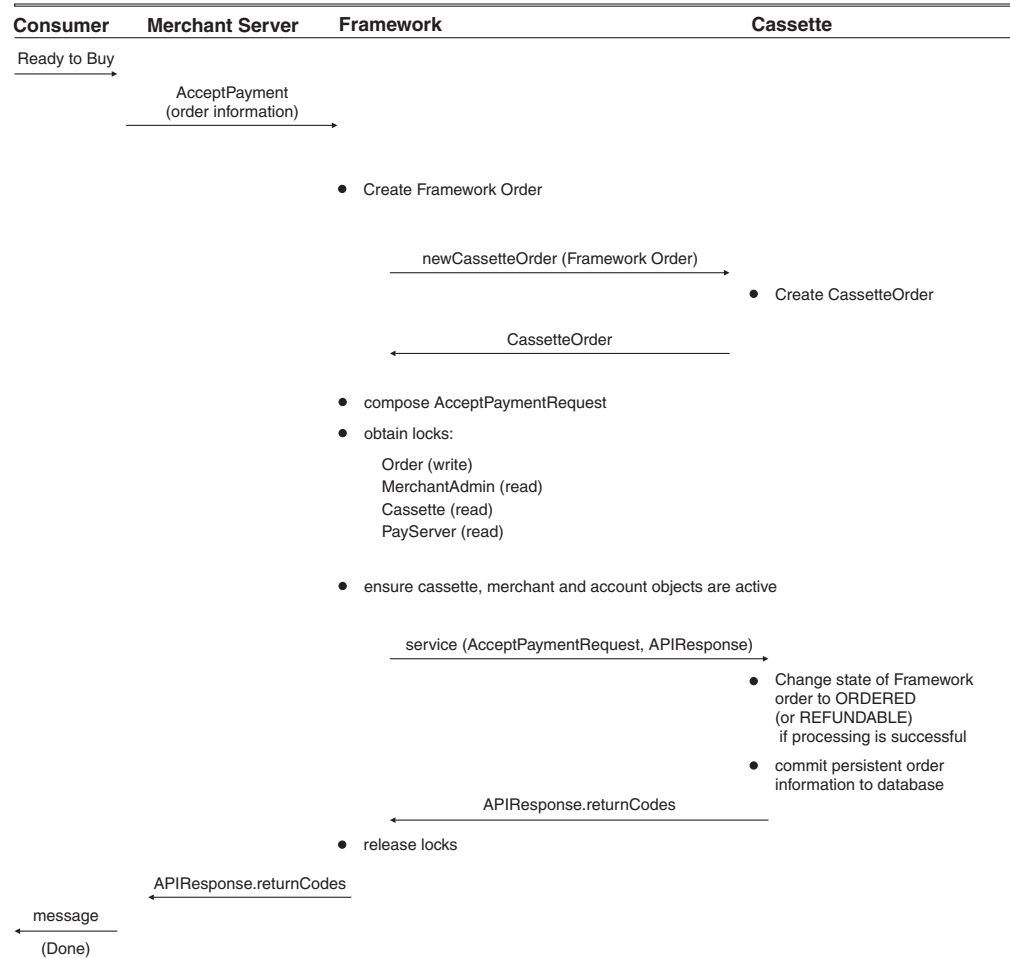


If the cassette supports independent credit, the cassette changes the state of the framework order to REFUNDABLE (rather than ORDERED) if processing is successful.

Completion of the ReceivePayment sequence (sending the message at the end) causes subsequent protocol message flows between the consumer (or some other agent) and the cassette. During these flows (before the order enters the ORDERED state), the cassette must fill in the correct account number in the generic order.

AcceptPayment API sequence

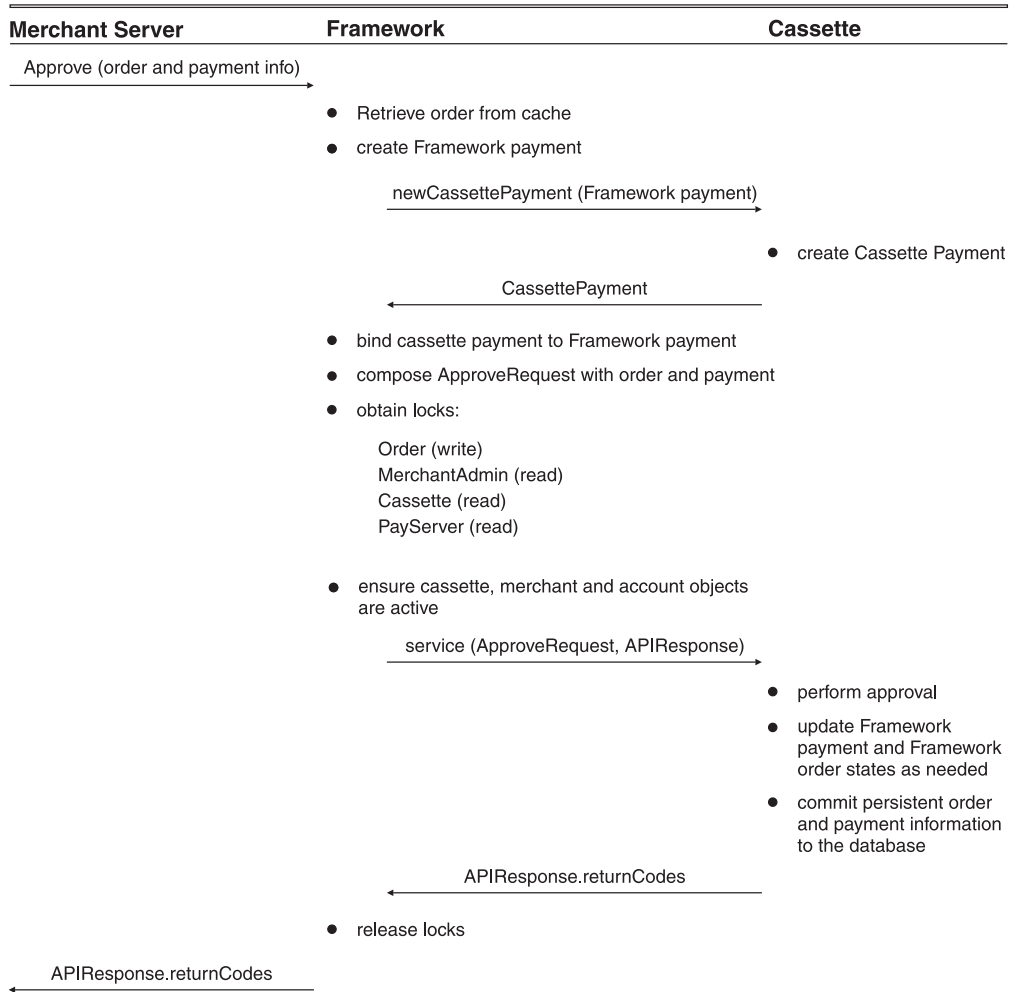
The AcceptPayment sequence diagram shows the interactions between the WebSphere Commerce Payments framework and cassette resulting from a AcceptPayment command from the merchant shopping software to WebSphere Commerce Payments.



If the cassette supports independent credit, the cassette changes the state of the framework order to REFUNDABLE (rather than ORDERED) if processing is successful.

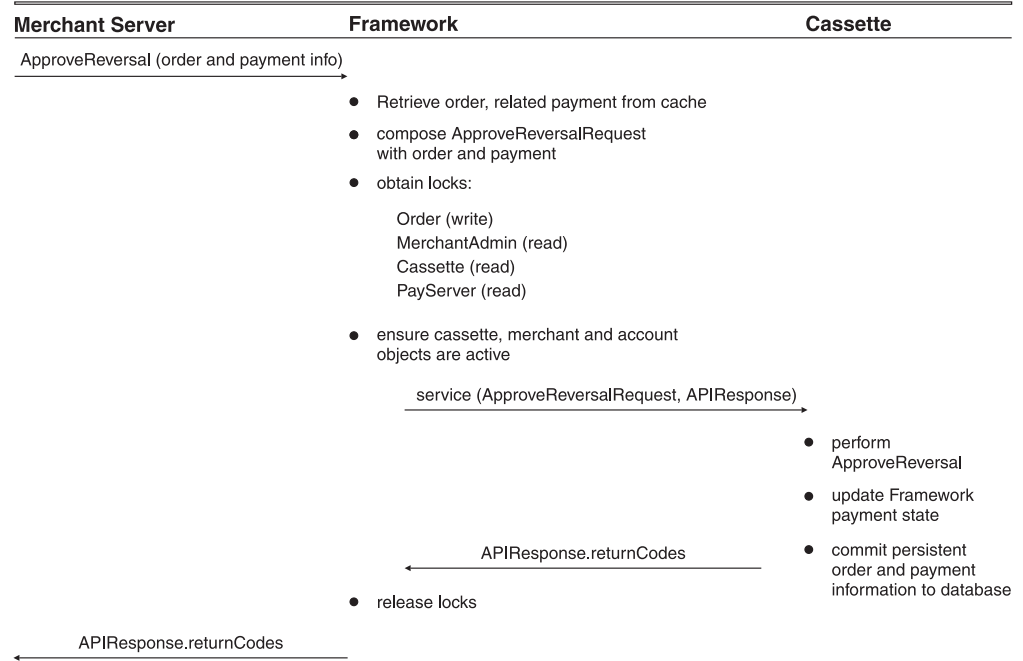
Approve API sequence

The Approve sequence diagram shows the interactions between WebSphere Commerce Payments framework and cassette resulting from an Approve command from the merchant shopping software to WebSphere Commerce Payments.



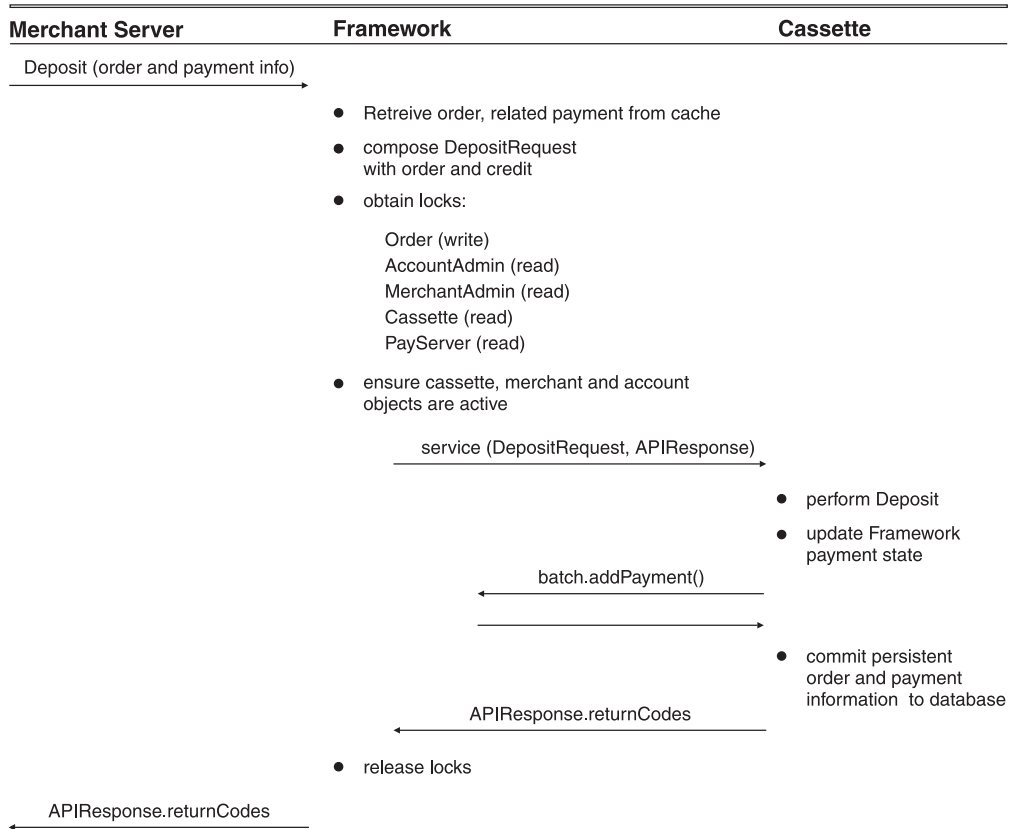
ApproveReversal API sequence

The ApproveReversal sequence diagram shows the interactions between the WebSphere Commerce Payments framework and cassette resulting from an ApproveReversal command from the merchant shopping software to WebSphere Commerce Payments.



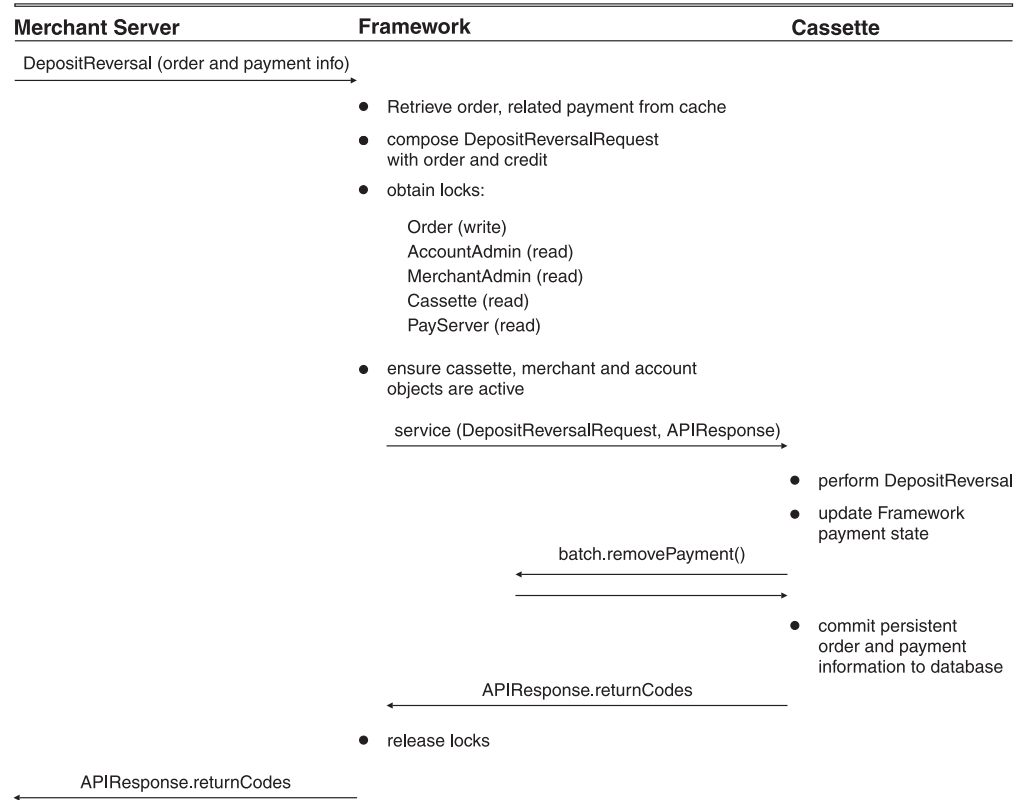
Deposit API sequence

The Deposit sequence diagram shows the interactions between the WebSphere Commerce Payments framework and cassette resulting from a `Deposit` command from the merchant shopping software to WebSphere Commerce Payments.



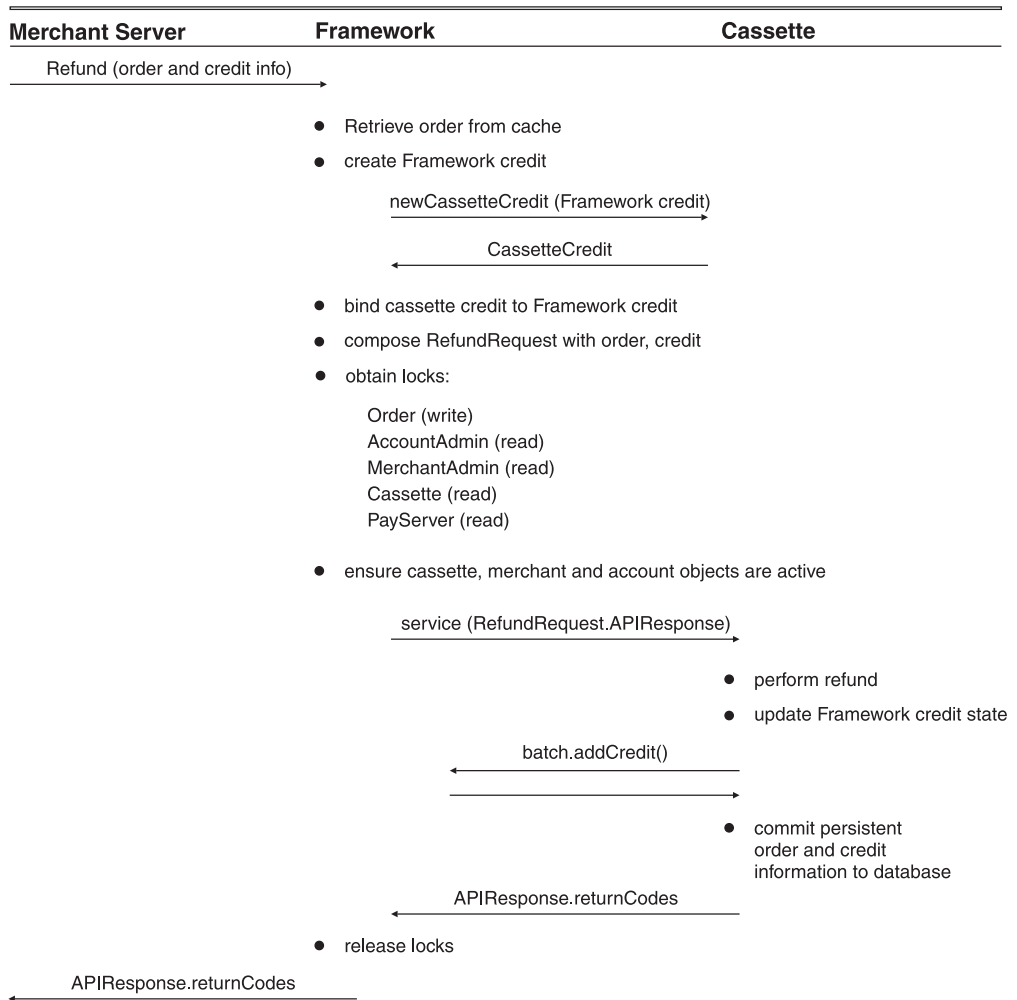
DepositReversal API sequence

The DepositReversal sequence diagram shows the interactions between the WebSphere Commerce Payments framework and cassette resulting from a DepositReversal command from the merchant shopping software to WebSphere Commerce Payments.



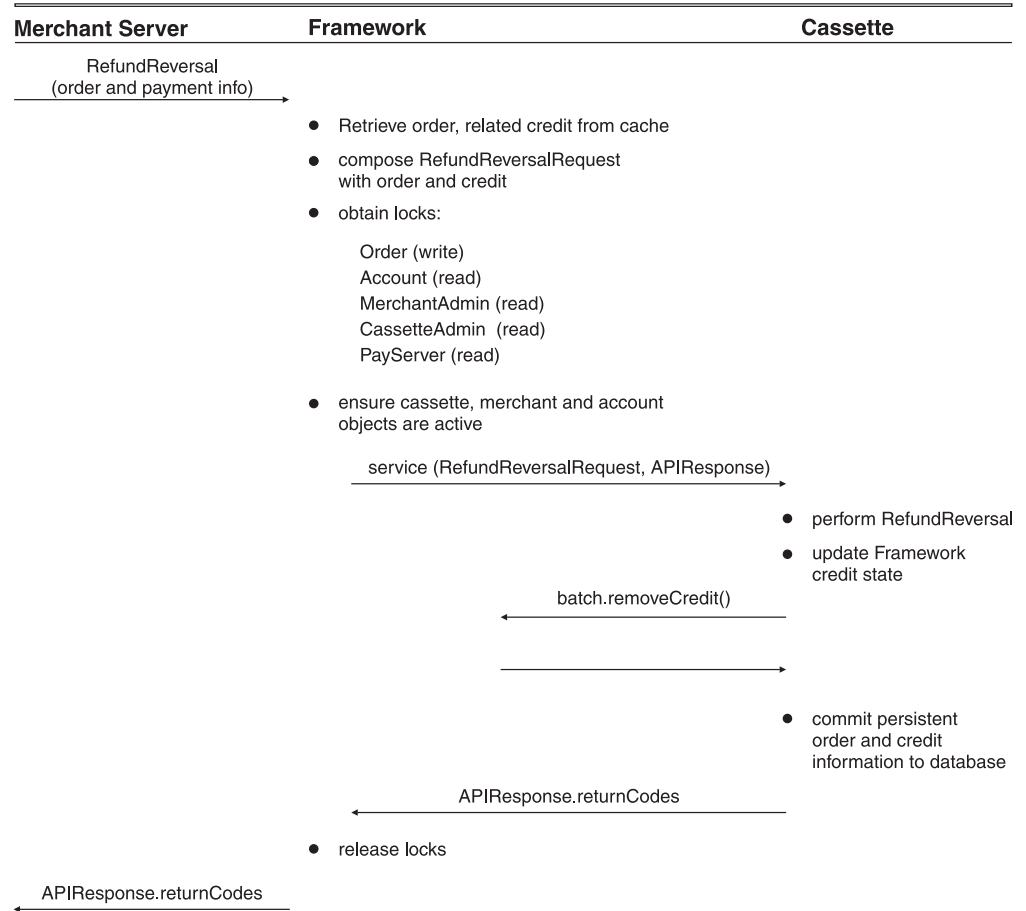
Refund API sequence

The Refund sequence diagram shows the interactions between the WebSphere Commerce Payments framework and cassette resulting from a Refund command from the merchant shopping software to WebSphere Commerce Payments.



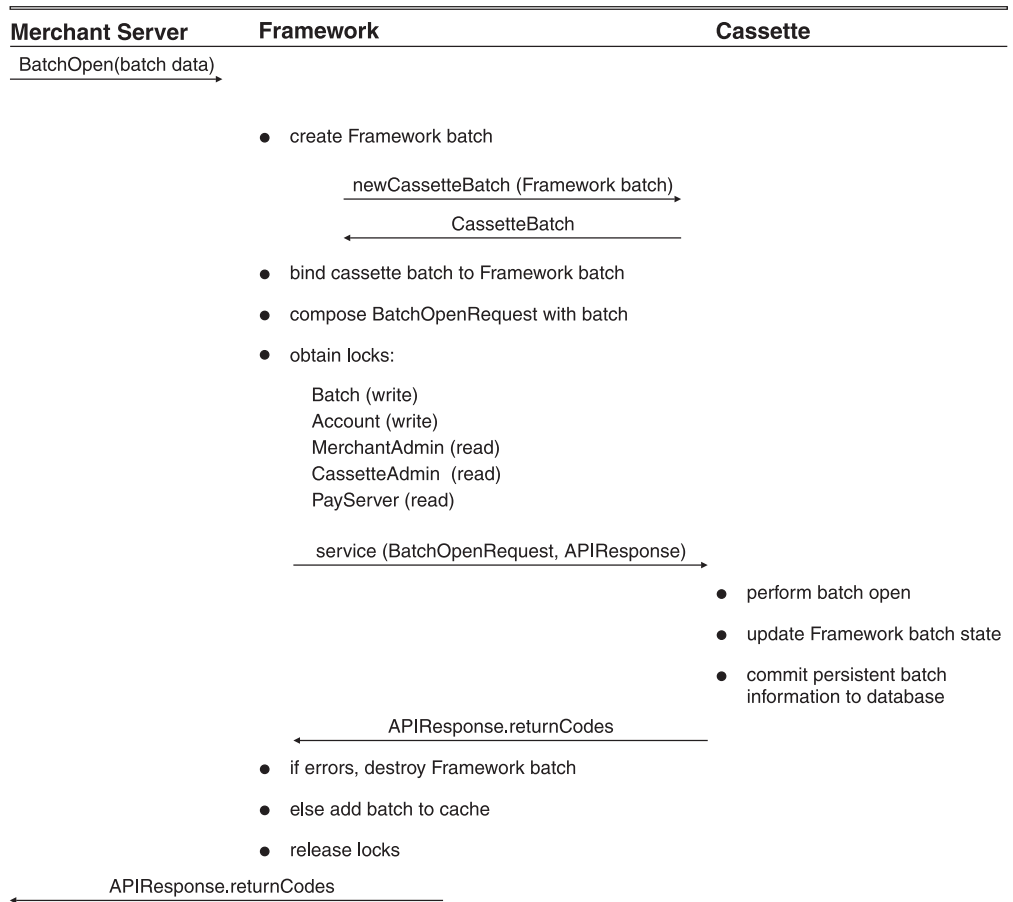
RefundReversal API sequence

The RefundReversal sequence diagram shows the interactions between the WebSphere Commerce Payments framework and cassette resulting from a RefundReversal command from the merchant shopping software to WebSphere Commerce Payments.



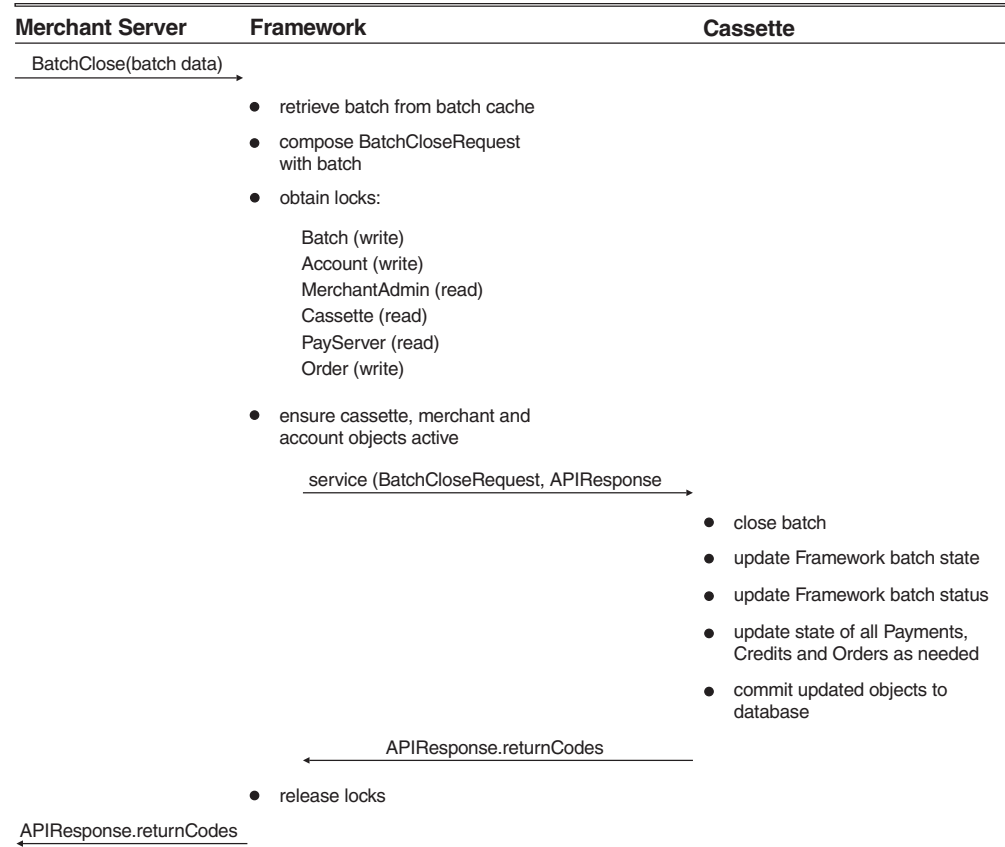
BatchOpen API sequence

The BatchOpen sequence diagram shows the interactions between the WebSphere Commerce Payments framework and cassette resulting from a BatchOpen command from the merchant shopping software to WebSphere Commerce Payments.



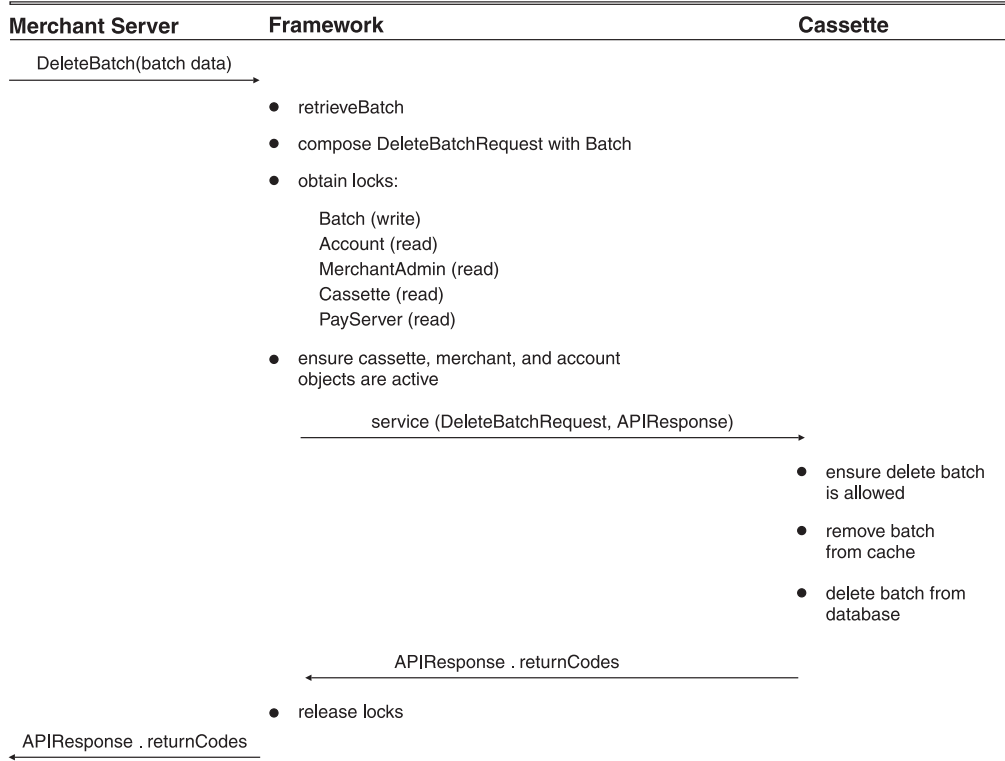
BatchClose API sequence

The BatchClose sequence diagram shows the interactions between the WebSphere Commerce Payments framework and cassette resulting from a BatchClose command from the merchant shopping software to WebSphere Commerce Payments.



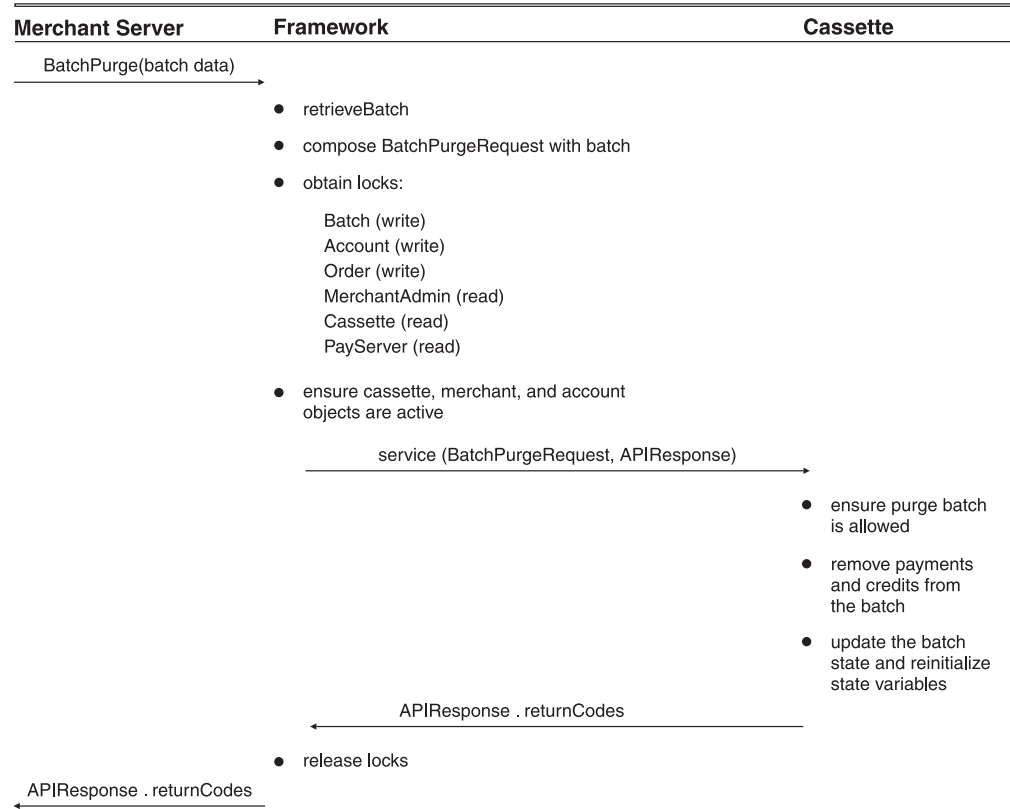
DeleteBatch API sequence

This DeleteBatch sequence diagram shows the interactions between the WebSphere Commerce Payments framework and cassette resulting from a DeleteBatch command from the merchant shopping software to WebSphere Commerce Payments.



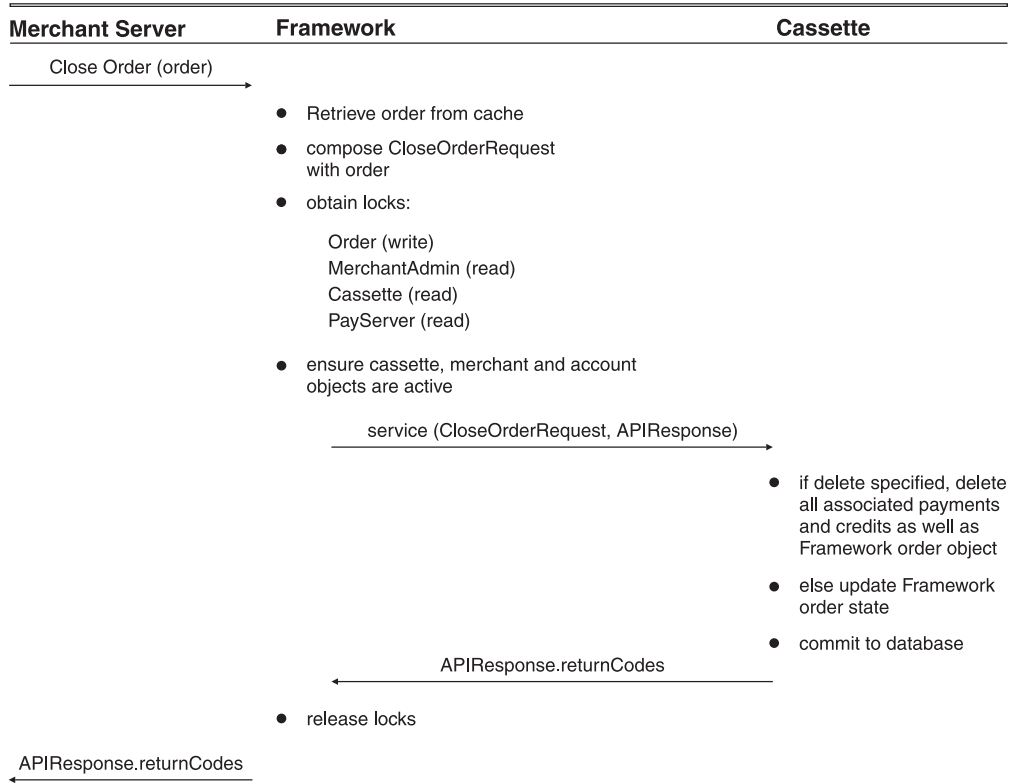
BatchPurge API sequence

This BatchPurge sequence diagram shows the interactions between the WebSphere Commerce Payments framework and cassette resulting from a BatchPurge command from the merchant shopping software to WebSphere Commerce Payments.



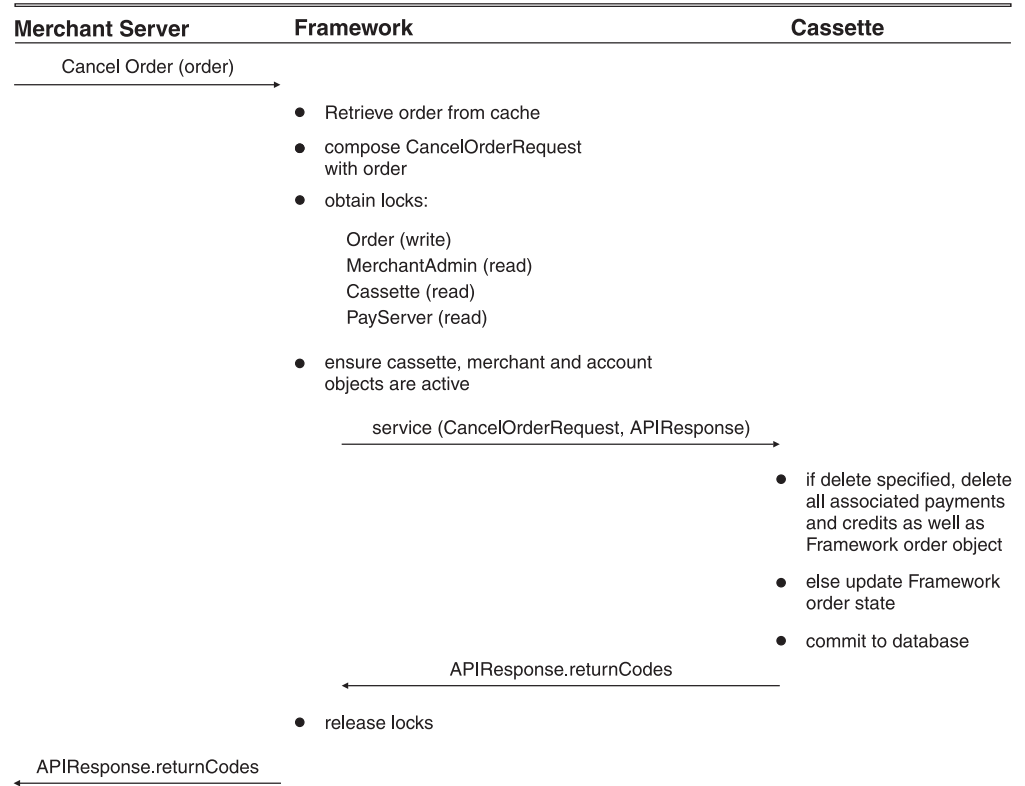
CloseOrder API control sequence

The CloseOrder sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette resulting from a CloseOrder command from the merchant shopping software to WebSphere Commerce Payments.



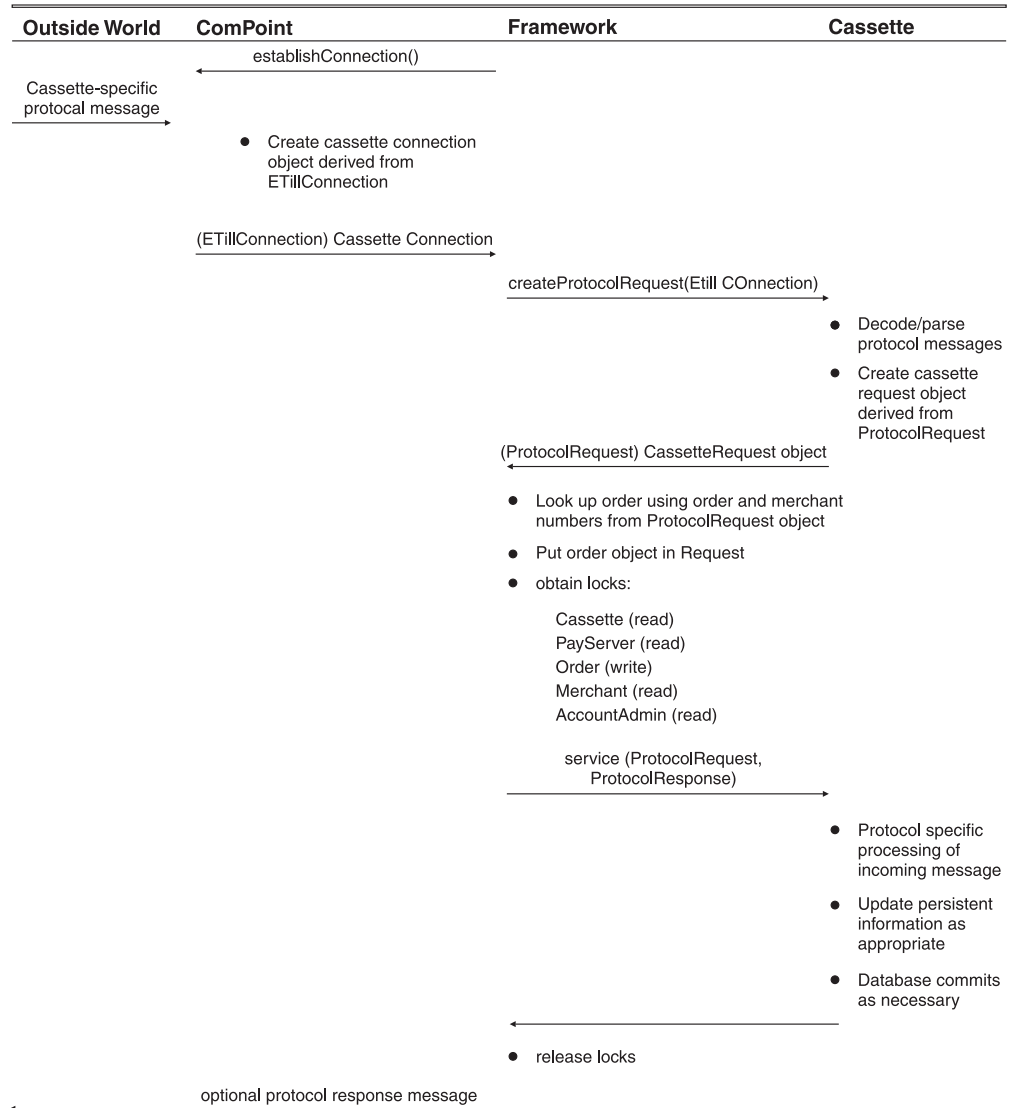
CancelOrder API sequence

The CancelOrder sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette resulting from a CancelOrder command from the merchant shopping software to WebSphere Commerce Payments.



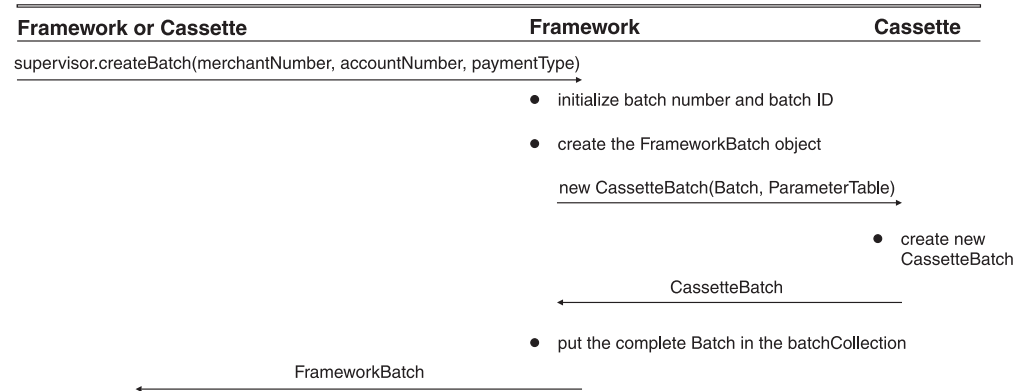
Protocol message API sequence

This sequence diagram shows the interaction between the WebSphere Commerce Payments framework and cassette when a protocol message specific to the cassette is received from the outside world.



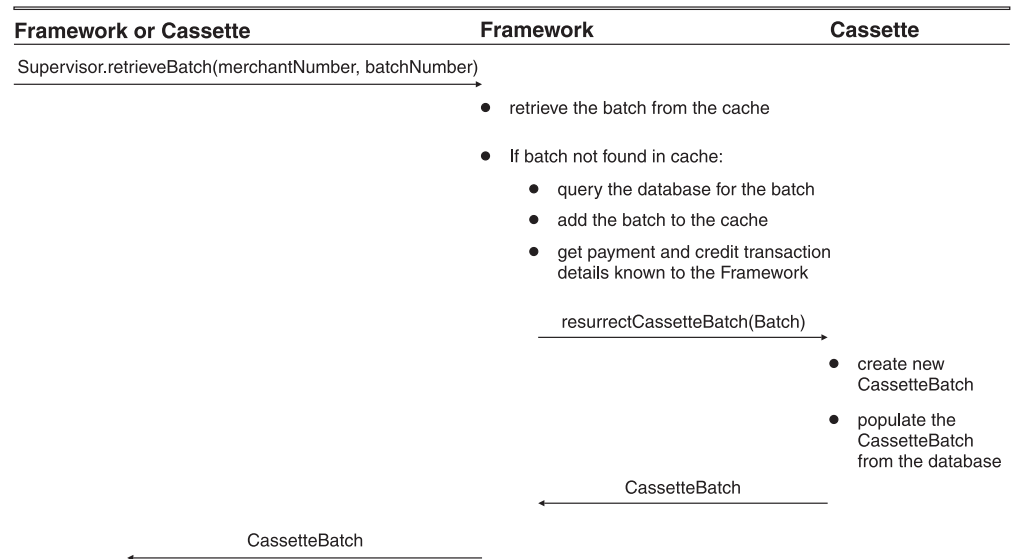
CreateBatch internal sequence

This sequence illustrates the internal flows required to create a new batch. A batch can be created by the framework or the cassette. For example, a cassette would create a new batch for a payment or credit that doesn't fit into an existing batch. This internal sequence can be invoked by either the framework (typically as the result of a BatchOpen command as shown in "BatchOpen API sequence" on page 78) or by the cassette when a batch must be created using the implicit style.



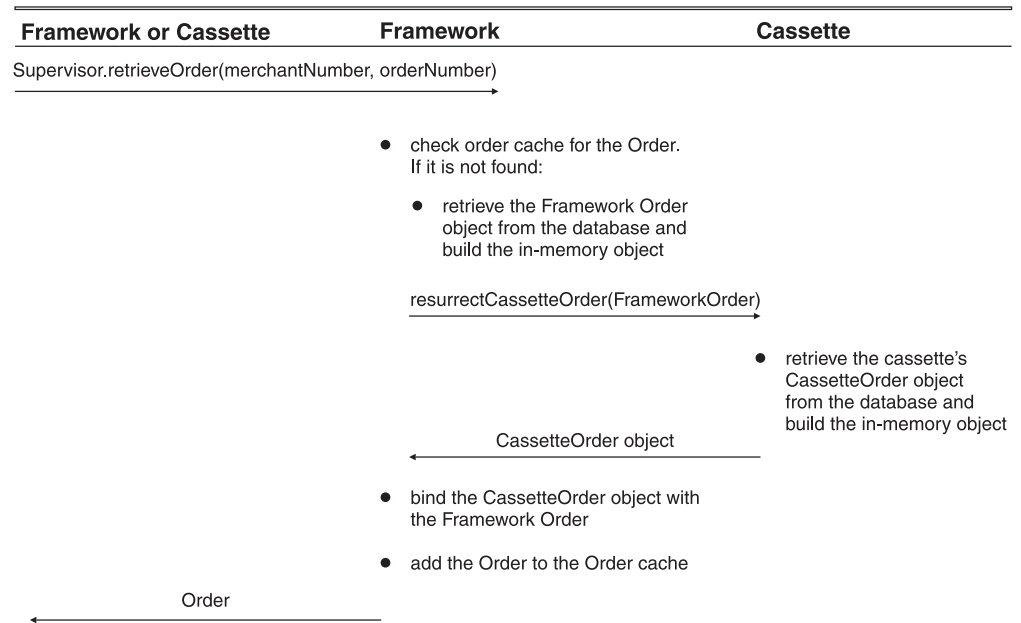
RetrieveBatch internal sequence

This sequence is invoked by the framework or the cassette to gain access to a given Batch object.



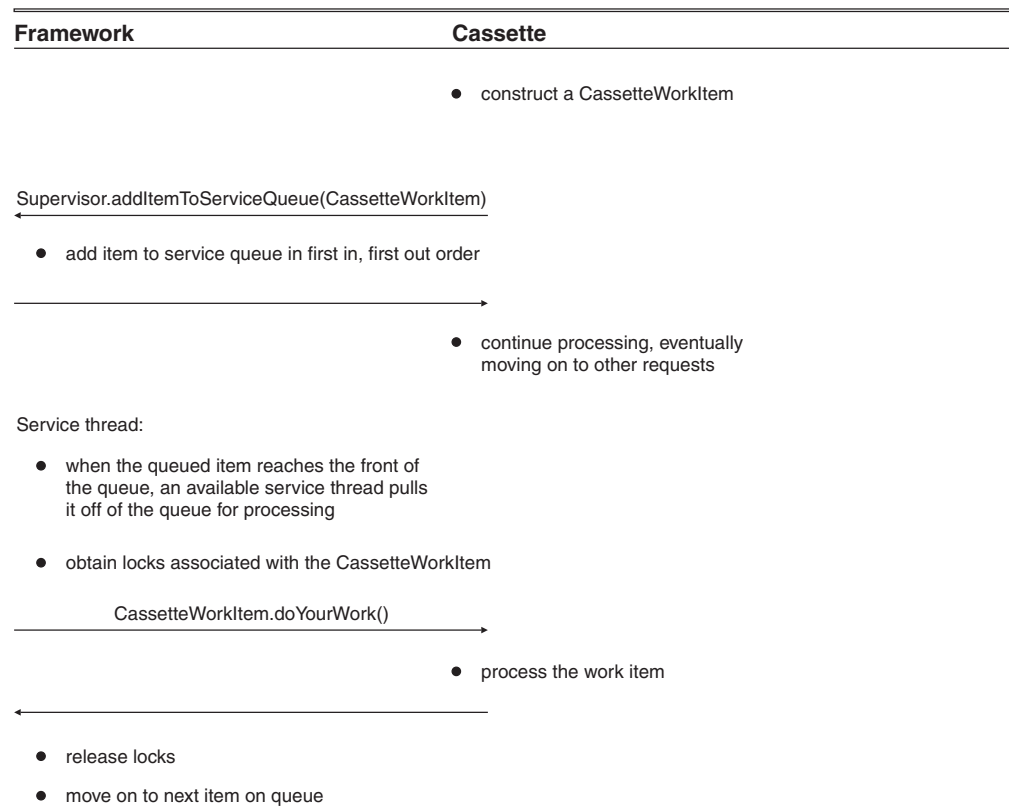
RetrieveOrder internal sequence

The RetrieveOrder internal sequence is invoked by the framework or the cassette to gain access to a given Order object.



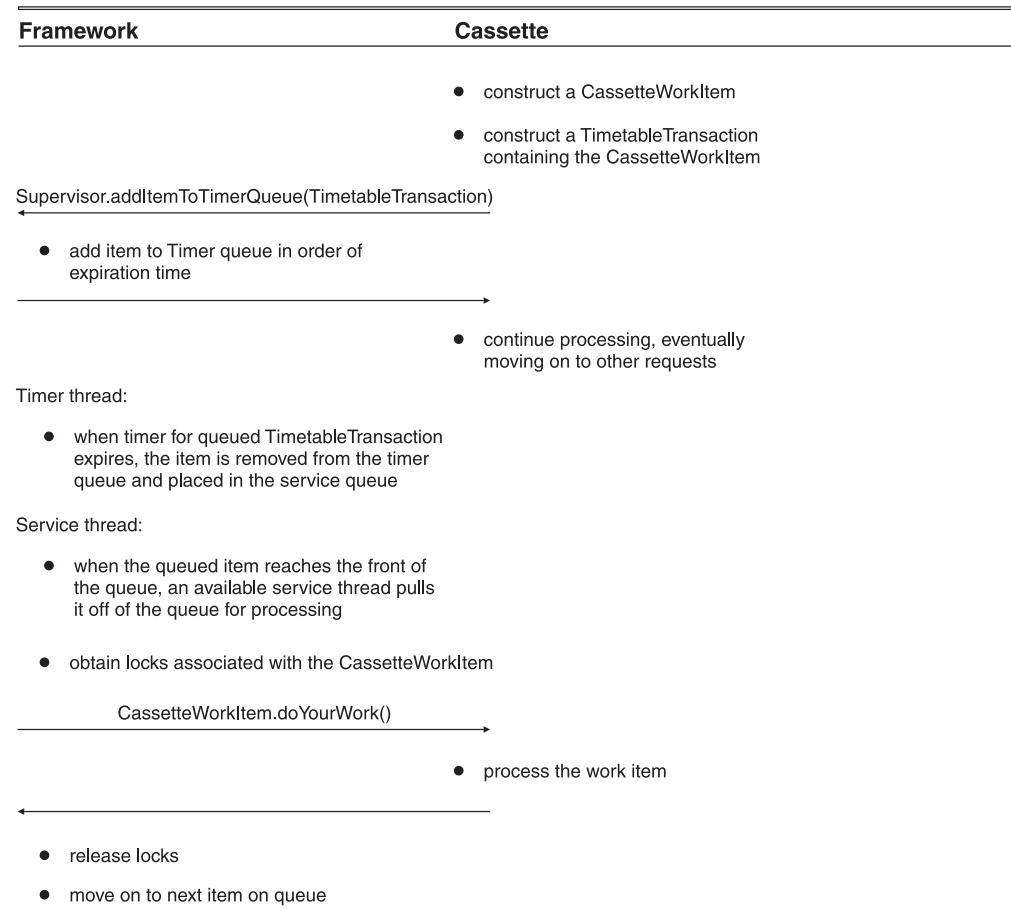
Service queue internal sequence

The Service Queue sequence is invoked by the cassette to schedule a work item to be executed on one of the framework's service threads.



Timer queue internal sequence

The Timer queue internal sequence is invoked by the cassette to schedule a work item to be executed on one of the framework's service threads after a specified amount of time.

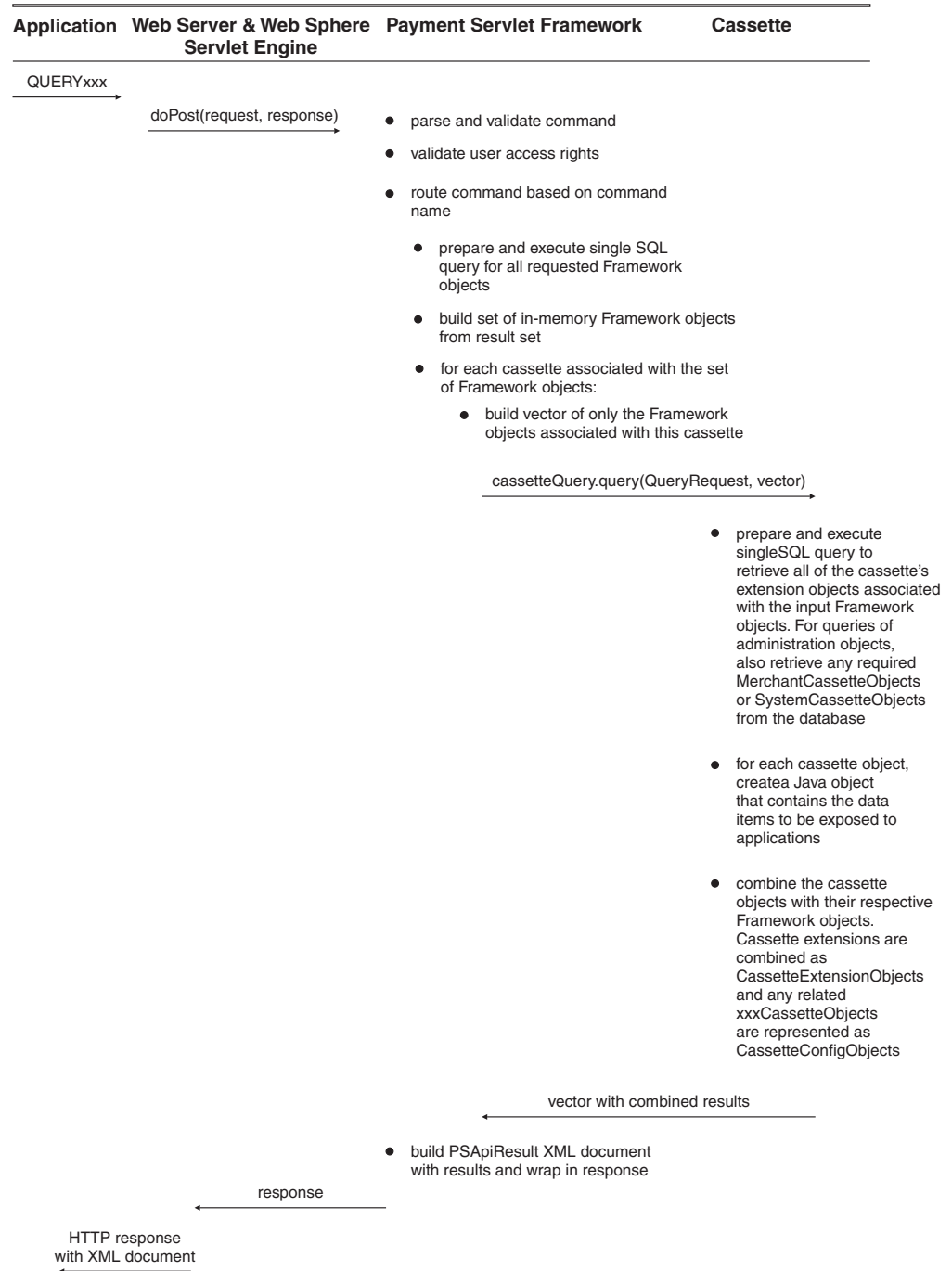


Query command processing

For all Query commands, the main interface between the framework and the cassette is the `CassetteQuery` object's `query` method.

Query API sequence

The Query sequence diagram shows the interactions between the WebSphere Commerce Payments framework and the cassette resulting from any of the Query API commands.



Responsibilities and services

Framework responsibilities include:

- “Access control”
- “Order and batch caching”
- “Threading” on page 92
- “Synchronization” on page 94
- “Parameter validation” on page 96
- “Background and timed operations” on page 99
- “Receiving protocol messages from the outside world” on page 102
- “Database access” on page 102
- “Event notification” on page 105
- “Error logging” on page 115
- “Debug tracing” on page 119

Access control

One of the key security features provided by WebSphere Commerce Payments for *hosted payment services* (a WebSphere Commerce Payments that supports many different merchants) is access control. The access control feature of the framework works in conjunction with WebSphere Application Server’s user authentication services to ensure that each merchant is only allowed to view or manipulate his or her own data.

Because all of the WebSphere Commerce Payments access control facilities operate transparently to the cassette, cassette writers do not have to worry about this issue. However, as new functionality is added to WebSphere Commerce Payments, access control services may be added to the cassette programming interface if it becomes necessary for cassettes to validate specific operations to the roles maintained for the requesting user or application. For example, if your cassette processes sensitive data, you may want to restrict who should view data returned in query command results. See “Protecting sensitive data” on page 29 for more information.

For more information on access control, see the *WebSphere Commerce Administration Guide*.

Order and batch caching

The framework ensures that there is only one in-memory object representing an Order and all associated data in existence at any given time. It does this by managing a cache of framework Order objects. Order objects are placed into this cache when they are first created and as a result of an internal call to the `Supervisor.retrieveOrder` method and they will remain in the cache until they are no longer being referenced by any cassettes. The framework tracks references internally, so it knows when it is safe to flush Orders from the cache without any explicit input from the cassette.

The framework also ensures that there is only one in-memory object representing each Batch by maintaining a cache of framework Batch objects. Batch objects are placed into the Batch cache when they are created and when the `Supervisor.retrieveBatch` method is called inside of WebSphere Commerce Payments. Batches remain in this cache as long as they are open. Batch objects contain lists of framework Payment and Credit objects. There are a number of abstract methods defined in `com.ibm.etill.framework.cassette.Cassette` to support

the framework's caching. These methods either create or reinstantiate (resurrect) the cassette-specific counterparts to the Order, Payment, Credit and Batch objects. It is important to note that the cassette **must not** create or resurrect any of these objects unless it is asked to do so by the framework.

The framework will ask the cassette to create new objects, as they are needed based on incoming API commands. The framework will ask the cassette to resurrect objects when operations on those objects are requested but the objects are not found in the cache. Cassettes should not attempt to maintain any of these objects in-memory across service calls. The framework will retrieve the required objects from the cache, or resurrect them using a call to the cassette. The framework will hand the necessary objects to the cassette using the service call. These objects may be safely used by the cassette until the service call returns but must not be held beyond the return of the service call.

Cassettes must be very careful to ensure that they do not inadvertently subvert the framework's ability to deliver on its guarantees concerning in-memory objects. Specifically, cassettes:

- **Should not** save references to these objects across service calls
- **Should not** pass references to these objects to another thread, including service threads
- **Should not** keep their own cache or lists of in-memory objects
- **Should not** make their own "copies" of any of these in-memory objects

Long-lived references are dangerous because the cassette has no way of knowing when cached objects might be flushed from the cache or when administrative objects might be deleted or replaced. Creating copies of the objects would violate the framework's guarantee of a single in-memory representation of the data and the associated synchronization of operations on payment data.

As mentioned above, the framework hands the objects necessary to process each service call to the cassette when the service call is made. However, if the cassette needs to access any objects that are not passed with the service request, it should always ask the framework for the in-memory objects. The framework provides all of the methods necessary to accomplish this.

Threading

Threading is an important part of the middleware server function provided by the WebSphere Commerce Payments framework. The framework provides all of the WebSphere Commerce Payments threads and controls the scheduling of cassette processing on those threads. By assuming control over the threading model, the framework ensures that it can change over time to improve performance and it protects the cassette writer from some of the complexities of writing a middleware server.

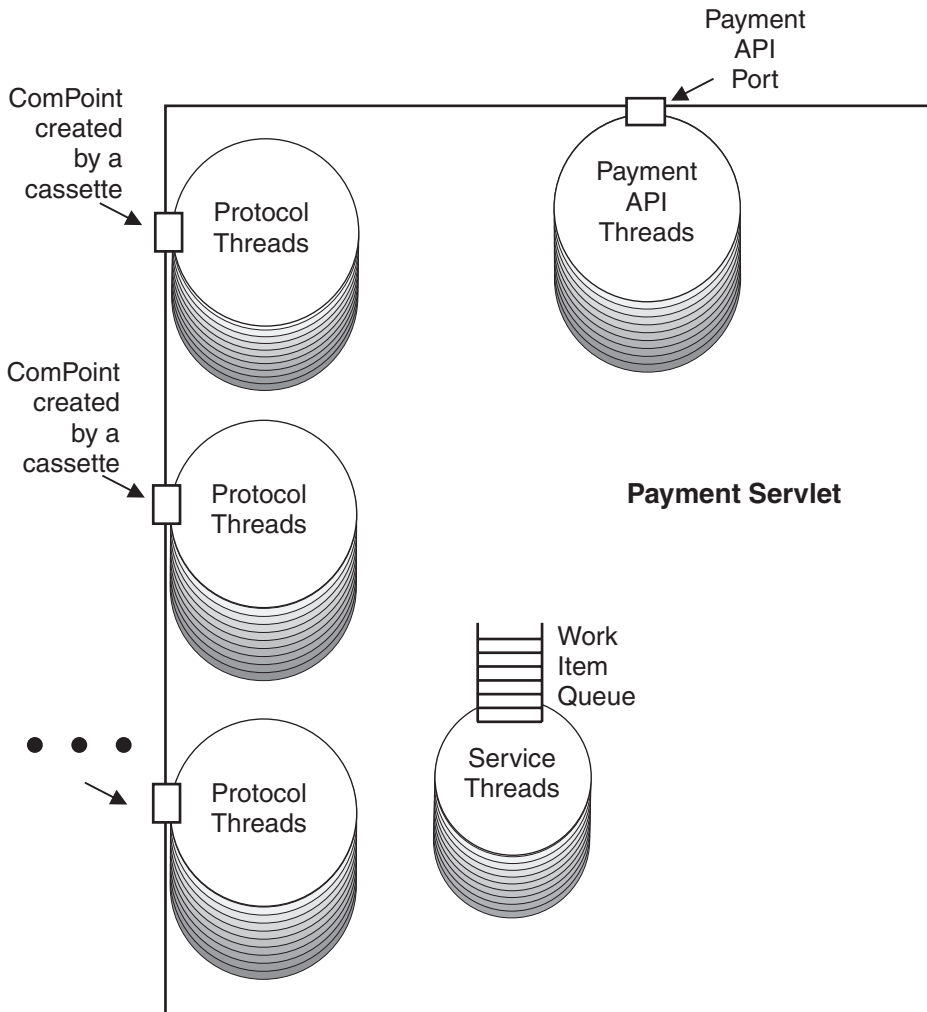
WebSphere Commerce Payments starts a fixed number of threads per cassette. The number of threads started is configurable, as described in the performance tuning parameters section of the *WebSphere Commerce Administration Guide*. The architecture of WebSphere Commerce Payments, however, ensures that processing can vary independently from the number of threads. As performance data becomes available, future versions of WebSphere Commerce Payments will be modified to provide improved performance by varying the number of threads, or even by

changing the thread model. A correctly-written cassette will benefit from these performance improvements and will not have to be modified if the thread model is changed.

Cassettes should **never create or start** a thread. This is very important! Violating this rule could thwart the framework's ability to provide performance improvements, synchronization, and guaranteed-unique, in-memory copies of data. Where cassettes need to move work to another thread, they should use the framework-provided service threads. The framework also allows cassettes to delay work until some future point in time, where it will run on a service thread. These capabilities are described in "Background and timed operations" on page 99.

Because the framework provides these services, any desire to start a thread probably indicates a misunderstanding of the Cassette interface and the framework functionality. Cassette function should be written with the basic assumption that any piece of function can run at any time on any thread. Another way to think about this is that any piece of cassette function can be running simultaneously with any other piece. Examples of a "piece of function" are a single call to the cassette's `service()` method by the framework and a single call to one of the cassette's caching support functions, such as `newCassetteOrder()`.

The synchronization, service thread, and timer thread sections in this guide provide related information.







-  = **Queue**
-  = **Thread**
-  = **Connections to the outside world**
-  = **The actual number of ComPoints depends on which cassette points are configured**

Figure 4. Payment Servlet thread model

Synchronization

The framework makes certain promises to the cassette when a service call is made. The first is that there is one, and only one, in-memory copy of the framework and cassette objects as defined in the framework payment data model. This promise results from the object caching provided by the framework and depends on the cassette's cooperation in following the caching rules.

The second promise is that no other thread will modify (or, if necessary, even access) those objects for the duration of the service call. This promise results from the framework's synchronization policy that also depends on the cassette's cooperation in following the caching rules. As long as the cassette follows the caching rules, all synchronization is done within the framework without requiring any action from the cassette.

The goal of the framework's synchronization policy is first to ensure data integrity and thread safety within the Payment Servlet and second to minimize the granularity of the locks in order to provide for the best possible concurrency between all of the threads in the Payment Servlet.

The framework's synchronization policy is based on a list of rules that is maintained with each request object. That is, each request object tells the framework which objects need to be locked, and which type of lock to obtain for each. Two different types of locks are supported:

Read lock

allows the holder to read data from the object with the guarantee that the values contained in the object will not change while the lock is held. Multiple threads may concurrently hold a read lock on an object. This is a direct analogy to a shared read-only lock on a file.

Write lock

allows the holder to modify the contents of the object with the guarantee that no other thread in the system is currently accessing the object. This is a direct analogy to an exclusive write lock on a file.

The list of objects locked for each request varies, but always includes some subset of the following:

Framework lock

controls access to the Payment Servlet itself. For all service requests, a read lock is obtained on this object to ensure that the Payment Servlet is not shut down before the cassette can finish processing the service call.

Cassette lock

controls access to the cassette. A read lock ensures that the cassette is not stopped before it can finish processing the service call. A write lock allows the cassette to modify its own runtime properties, start itself up or shut itself down without adversely impacting any other request that is being processed by the cassette.

MerchantAdmin lock

controls access to the merchant. A read lock ensures that the merchant is not disabled or modified before the cassette can finish processing the service call. A write lock allows the cassette to modify its merchant-specific configuration information without adversely impacting any other request that is being processed anywhere within the Payment Servlet.

Account lock

controls access to the Account. A read lock ensures that the Batch with which the service call is associated will not begin CLOSE processing before the cassette can finish processing the service call. If communication with an external entity (like a financial institution) is required, this lock also ensures that messages can be exchanged with that entity in parallel with other such messages from other Payment Servlet threads. A write lock

prevents any other activity from taking place on the Account object. This type of lock is obtained when a batch is being opened or closed.

Order lock

controls access to a given Order and all of its associated objects (that is, framework Payments and Credits and cassette's Order, Payment and Credit objects). A read lock ensures that the Order or its associated objects will not be modified before the cassette can finish processing the service call. A write lock allows the cassette to modify the Order or any of its associated objects without adversely impacting any other request that is being processed anywhere within the Payment Servlet.

Batch lock

controls access to a given Batch object. A read lock ensures that the Batch object will not be modified before the cassette can finish processing the service call. A write lock allows the cassette to modify the Batch without adversely impacting any other request that is being processed anywhere within the Payment Servlet.

For a detailed description of the locking performed for each request, see the Javadoc for `OrderRequest`, `ProtocolRequest`, and `CassetteWorkItem`. The class description for each of these (except `CassetteWorkItem` - see method descriptions) contains a table that presents this information.

Parameter validation

The framework provides cassette developers with the ability to provide the parsing syntax rules for the cassette's protocol data parameters. These rules are supplied by the cassette through a Name parameter validation table. A cassette's Name parameter validation table should be built during cassette initialization. The table should contain the parsing rules, expressed in terms of the various subclasses of `com.ibm.etill.framework.payapi.validator.SimpleValidator`, for every protocol data parameter that the cassette supports.

Note: Previous versions of WebSphere Commerce Payments used the `ParameterValidationItem` class and its associated subclasses. If you used this approach, you should now use the `Validator` classes.

Note that only one such table should exist for the cassette.

Each time the framework parses an incoming command that is directed toward the cassette, it will call the following method on the cassette's `com.ibm.etill.framework.cassette.Cassette` object to access the cassette's validation table:

```
public Hashtable getParameterValidationTable()
```

After accessing the cassette's table, the framework parses the protocol data parameters according to the rule defined in the table.

The framework has a default parameter validation table that it will use if this method is not overridden by the cassette developer. The default table is empty, thus the default behavior is to not perform any parameter validation. If a cassette developer wishes to do parameter validation in this way then a fully-populated, cassette-defined `Hashtable` should be returned in the method `getParameterValidationTable()` that is called by the framework whenever an API command is received.

The cassette must create its own parameter validation Hashtable and populate it with key-value pairs that correspond to each parameter it wishes to do validation on. The key is a protocol data parameter keyword on an API command and the value for that key is the appropriate subclass of the SimpleValidator class (for example, StringValidator). The `com.ibm.etill.framework.payapi.validation.SimpleValidator` class is the abstract base class for all validation item classes. At a minimum, each of the validation item classes take this information in the constructor:

- The parameter name
- The return code to return in the API response if parameter fails validation
- A boolean that indicates if null parameter values are allowed.

The following validation item classes are provided by the framework (and are subclasses of `ParameterValidatorItem`):

Aliasing

allows you to define an alias for another protocol data keyword. This is especially useful for defining keywords that are accessible from applications that are coded to the Payment Server Version 1.2 compatibility interface, which limits protocol data keywords to 16 characters (including the '\$'). In version 3.1, the `AliasValidationItem` class is no longer needed. Everything can be accomplished through the validator classes. For example, to allow `$AVS.STREETADDR` to act as an alias for `$AVS.STREETADDRESS`, you would code the following:

```

IPParameterValidator val = // $AVS.STREETADDRESS
    new StringValidator(
        PD_AVS_STREETADDRESS,
        RC_CASSETTE_AVS_STREETADDRESS, // 1053
        false, // No null values
        1, // Minimum length
        24, // Maximum length
        ISO8859_1 // ASCII (for now)
    );
validationTable.put( PD_AVS_STREETADDRESS, val ); // $AVS.STREETADDRESS
validationTable.put( PD_AVS_STREETADDR, val ); // $AVS.STREETADDR alias

```

BooleanValidator

does validation for boolean parameters.

ByteArrayValidator

does validation for byte array parameters. In addition to the required parameters, a maximum length parameter must also be specified on the constructor.

IntegerValidator

does validation for integer parameters. In addition to the required parameters, a minimum value and maximum value must also be specified on the constructor.

LongValidator

does validation for long parameters. In addition to the required parameters, a minimum value and maximum value must also be specified on the constructor.

NumericStringValidator

does validation for numeric string parameters. This class is a subclass of `StringValidator`. A numeric string parameter is a `String` parameter that always represents a number (for example, 9876). In addition to the required parameters, a minimum length, maximum length, minimum value and maximum value must also be specified on the constructor.

NumericTokenValidator

does validation for numeric token parameters. This class is a subclass of `StringValidator`. A numeric token parameter is a `String` parameter that represents a number with the following constraints: a null value is not allowed, the numeric string cannot contain leading zeros, the numeric string cannot be negative, the string has a minimum length of 1 and a maximum length of 9. In addition to the required parameters, a minimum value and maximum value must also be specified on the constructor.

NumericTokensValidator

does validation for a `Vector` of numeric token parameters. This class is a subclass of `StringValidator`. A numeric token parameter is a `String` parameter that represents a number with the following constraints: a null value is not allowed, the numeric string cannot contain leading zeros, the numeric string cannot be negative, the string has a minimum length of 1 and a maximum length of 9. In addition to the required parameters, a minimum value and maximum value must also be specified on the constructor.

PathValidator

does validation for path parameters. The validation will ensure that the specified path is a valid directory name with write permission. In addition to the required parameters, a maximum length must also be specified on the constructor.

RestrictedStringValidator

does validation for string parameters that have character(s) that are not allowed to be in the string. This class is a subclass of `StringValidator`. In addition to the required parameters, a minimum length, a maximum length, the encoding to use, and a string that indicates characters that must be excluded from the string must also be specified on the constructor.

StringValidator

does validation for string parameters. In addition to the required parameters, a minimum length, maximum length, and the encoding to use must also be specified on the constructor.

TimestampValidator

does validation for time stamp parameters. The passed in `String` parameter is converted to a `long` (using `8859_1` encoding) and then converted into a `java.sql.Timestamp` object using the public `Timestamp(long time)` constructor. The cassette developer can use these framework classes as is, or can create their own parameter validation classes by creating a subclass of the appropriate framework validation item class.

Each validation item class has a method called:

```
protected Object validateAndInsertValue(string value, ParameterTable resultTable)
```

that does the validation of the parameter and, if valid, inserts it into a `ParameterTable`. If the parameter is not valid, then an `ETillAbortOperation` exception is thrown, using the return code that was specified in the constructor of the `ValidationItem` class. Thus, the framework will detect those validation errors that the cassette specified in the parameter validation table before control is ever passed to the cassette. If all framework level parameter validation succeeds, then the populated `ParameterTable` is passed from the framework to the cassette when an API command needs to be processed. The `ParameterTable` contains key-value pairs, where the key is the parameter name and the value is the value of the protocol data that was passed in the API command.

Examples:

```
public class SampleCassette extends Cassette
{
    //-----
    // Constants
    //-----
    public static final String PD_ACCOUNTNUMBER = "$ACCOUNTNUMBER";
    public static final String PD_ORDERFIELD1 = "$ORDERFIELD1";
    public static final String PD_ORDERFIELD2 = "$ORDERFIELD2";
    public static final String PD_ORDERFIELD3 = "$ORDERFIELD3";
    public static final String PD_ORDERFIELD4 = "$ORDERFIELD4";
    public static final String PD_ORDERFIELD5 = "$ORDERFIELD5";
    public static final String PD_ORDERFIELD6 = "$ORDERFIELD6";
    public static final String PD_ORDERFIELD6_ALIAS = "$ORDERFIELD6_ALIAS";

    public static final short RC_ACCOUNTNUMBER = (short) 10000;
    public static final short RC_ORDERFIELD1 = (short) 10001;
    public static final short RC_ORDERFIELD2 = (short) 10002;
    public static final short RC_ORDERFIELD3 = (short) 10003;
    public static final short RC_ORDERFIELD4 = (short) 10004;
    public static final short RC_ORDERFIELD5 = (short) 10005;
    public static final short RC_ORDERFIELD6 = (short) 10006;

    //-----
    // Parameter validation table
    //-----
    private Hashtable typeMapping = new Hashtable();

    //-----
    // Constructor
    //-----
    public SampleCassette()
    {
        // Initialize the parameter mapping hashtable.
        typeMapping.put(PD_ACCOUNTNUMBER,
            new NumericTokenValidator(PD_ACCOUNTNUMBER,RC_ACCOUNTNUMBER));
        typeMapping.put(PD_ORDERFIELD1, new StringValidator(PD_ORDERFIELD1,
            RC_ORDERFIELD1, false, 1, 32, "UTF8"));
        typeMapping.put(PD_ORDERFIELD2, new IntegerValidator(PD_ORDERFIELD2,
            RC_ORDERFIELD2, false, new Integer(0), null));
        typeMapping.put(PD_ORDERFIELD3, new TimestampValidator(PD_ORDERFIELD3,
            RC_ORDERFIELD3, false));
        typeMapping.put(PD_ORDERFIELD4, new BooleanValidator(PD_ORDERFIELD4,
            RC_ORDERFIELD4, false));
        typeMapping.put(PD_ORDERFIELD5, new ByteArrayValidator(PD_ORDERFIELD5,
            RC_ORDERFIELD5, false, new Integer(32000)));
        typeMapping.put(PD_ORDERFIELD6, new StringValidator(PD_ORDERFIELD6,
            RC_ORDERFIELD6, false, 1, 32, "8859_1"));
        typeMapping.put(PD_ORDERFIELD6_ALIAS, new AliasValidator(PD_ORDERFIELD6_ALIAS,
            PD_ORDERFIELD6,));
    }

    //-----
    // Return parameter validation table to framework
    //-----
    public Hashtable getParameterValidationTable() { return typeMapping;}
}
```

Background and timed operations

The framework maintains a timer thread and a pool of service threads that can be used by cassettes to schedule future work and to off-load work from the running thread. There is a single service queue that is served by the pool of service threads. The service queue is a queue of objects that implement the

`com.ibm.etill.framework.supervisor.WorkItem` interface. Service threads take one of these objects off the queue and call `WorkItem.doYourWork()`

There is a single timer queue served by a single timer thread. The timer queue is a queue of `TimeableTransaction` objects (that is, timeable `WorkItems`). The timer thread removes `TimeableTransactions` from the queue when their waiting period has expired. The `WorkItem` is removed from the `TimeableTransaction` and queued to the service queue where the work will actually be performed by one of the Service threads. All objects in the service queue must implement the `WorkItem` interface. There is no synchronization performed when `WorkItems` are executed on a Service Thread unless the `WorkItem` implementation itself ensures data integrity and thread safety.

The framework defines a `WorkItem` that is useful for cassettes, called `CassetteWorkItem` (`com.ibm.etill.framework.cassette.CassetteWorkItem`). `CassetteWorkItems` must be used very carefully since they represent new entry points into the cassette. `CassetteWorkItems` are essentially internally-generated requests. The issues regarding assurance of object uniqueness (one in-memory representation of the data) and synchronization are similar to those encountered for externally-generated events, such as API and protocol requests. As with any other subclass of the `CassetteRequest` class, each `CassetteWorkItem` contains a list of locks that need to be obtained before asking the cassette to process the request. This allows synchronization to occur, thus ensuring data integrity and thread safety for the `CassetteWorkItem`. This list may be formed in one of three ways (`CassetteWorkItem` has a different constructor for each):

1. If the `CassetteWorkItem` is constructed based on an existing instance of a `CassetteRequest` object, then the list of locks will be copied from that object.
2. If the `CassetteWorkItem` is constructed only with a request token, then the list of locks will consist of:
 - A Write lock on the `CassetteAdmin` object
 - A Read lock on the `PayServer` object

Note that these are rather coarse-grained locks (all other activity under the Cassette will be blocked during the eventual processing of this request).

3. If the `CassetteWorkItem` is constructed with a request token and a merchant number, then the list of locks will consist of:
 - A Write lock on the `MerchantAdmin` object
 - A Read lock on the `CassetteAdmin` object
 - A Read lock on the `PayServer` object

Note that these are rather coarse-grained locks. All other activity under the Merchant will be blocked during the eventual processing of this request.

What this means is that for most `CassetteWorkItems` (those constructed based on `APIRequest` objects), the framework will obtain the same set of locks when processing the workitem as it did for the original API request object.

Finally, it is important for cassettes to remove `CassetteWorkItem` objects that are pending on the framework's Service Queue or Timer Queue when the administrative object with which the work item is associated is stopped (disabled). For example, assume this scenario:

- While processing a Deposit command, a cassette builds a `CassetteWorkItem` based on the input `PaymentTransactionRequestItem` and adds this work item to the Timer Queue by calling `Supervisor.addItemToTimerQueue`, requesting a timer interval of 15 minutes.
- Three minutes later, the `AccountAdmin` object under which the Deposit command was being processed is disabled by the merchant administrator.

In this example, when the cassette's service method is called with the `STOP_ACCOUNT_TOKEN`, it must remove the `CassetteWorkItem` from the Timer queue by calling `Supervisor.removeItemFromTimerQueue`. Leaving work items on either of these queues after stopping the associated administrative object will cause an error to occur when the framework attempts to process them.

This example shows how to use a `CassetteWorkItem` and the framework's timer queue to write a record to the trace files every ten minutes:

```
//-----
// An example of putting a debug trace entry in the trace file every
// ten minutes.
//-----

//-----
// Code implemented by a cassette writer to add a work item.
//-----

...

// Set the first time for this work item to execute in 10 minutes from now.
long timeBetweenWorkItems = 1000 * 60 * 10;
long nextWorkItem = System.currentTimeMillis() + timeBetweenWorkItems;

// Create a TimeableTransaction and add it to the timer queue.
TimeableTransaction timeableTrx = new TimeableTransaction(this,nextWorkItem);
Supervisor.addItemToTimerQueue(timeableTrx);

...

//-----
// Work item class implemented by a cassette writer.
//-----

public class ExampleWorkItem extends CassetteWorkItem
{
    private void queueWorkItem()
    {
        // set the time for this work item to execute again in 10 minutes from now.
        long timeBetweenWorkItems = 1000 * 60 * 10;
        long nextWorkItem = System.currentTimeMillis() + timeBetweenWorkItems;

        // Create a TimeableTransaction and add it to the timer queue.
        TimeableTransaction timeableTrx = new TimeableTransaction(this,nextWorkItem);
        Supervisor.addItemToTimerQueue(timeableTrx);
    }

    // The method necessary to implement when extending CassetteWorkItem.
    // This is the method called when this WorkItem gets executed.
    public void doYourWork()
    {
        // Do the necessary work.
        Trace.traceDebug("CASSET", "ExampleWorkItem.doYourWork");
    }
}
```

```

        //put the item back on the queue.
        queueWorkItem();
    }
}

```

Receiving protocol messages from the outside world

The framework provides a communications hierarchy that must be subclassed, or used directly, to create Java classes that know how to send and receive your payment protocol's protocol messages. `ComPoint` is a Java interface that represents the "listening port" to the outside world. In Java Socket terms, a `ComPoint` would be a `ServerSocket`. In e-mail terms, this could be an e-mail acceptor. In file terms, this could be an object that watches for new files in a particular location.

`ETillConnection` is an interface that represents a single connection received by the `ComPoint`. In Java Socket terms, an `ETillConnection` would be a `Socket`. In e-mail terms, this would represent a single piece of e-mail. In file terms, this would be a single file.

To receive payment-protocol specific messages, threads must be dedicated to listening and incoming messages must be parsed. The framework provides the dedicated threads. Cassettes parse incoming messages, using objects of types derived from `ComPoint` and `ETillConnection` and defined by the cassette. The actual mechanics of listening (socket, e-mail, change to a file, etc.) are known only by the cassette. The cassette provides `ComPoints` that the framework plugs together with `ProtocolThreads` to provide the function of receiving protocol messages from the outside world.

Database access

The framework provides several very flexible and powerful database mechanisms:

- The *Archivable* interface allows cassettes to enable any of its objects for storage and retrieval in the WebSphere Commerce Payments database.

Archivable defines methods through which the object can be created, updated and deleted in the database. These methods must include the appropriate JDBC calls to perform the requested function. When a commit point is reached, the thread's `CommitPoint` object will call these methods as appropriate.

All of the framework's financial and administrative objects implement the *Archivable* interface, because they are all persistent objects. In addition, the primary cassette object interfaces `CassetteOrder`, `CassetteTransaction`, and `CassetteBatch`. All implement *Archivable*. However, *any* class that you decide should be represented in its own database table may implement *Archivable*. For more detail, see `com.ibm.etill.framework.archive.Archivable`.

- The *Restorable* interface allows cassettes to enable any of its objects to restore itself from data in the WebSphere Commerce Payments database. *Restorable* defines the method `public void restoreRecord()` that would be implemented by the cassette developer for any class that implements the *Restorable* interface. This method must include the appropriate JDBC calls to obtain the record to be restored from the database and populate the corresponding object with the data obtained from the database. When a commit point is reached, the thread's `CommitPoint` object will call this method as appropriate.

For more information, see `com.ibm.etill.framework.archive.Restorable`.

- The thread's `CommitPoint` collects objects to be added to, updated in, restored from, and deleted from the WebSphere Commerce Payments database and later executes those operations within a single atomic commit point.

The framework ensures that each thread always has an open `CommitPoint` object through which database operations can be performed. This object is accessed through the `Supervisor.getThreadCommitPoint` method.

A `CommitPoint` provides methods to add `Archivable` objects to any of three lists of operations to be performed the next time the `CommitPoint` is instructed to perform a commit operation:

- A list of objects to be added to the WebSphere Commerce Payments database
- A list of objects to be updated in the WebSphere Commerce Payments database
- A list of objects to be deleted from the WebSphere Commerce Payments database

`CommitPoint` also provides a `commit` method. When this is called, each of the lists is traversed. First, some optimizations are performed to minimize the amount of JDBC calls. For example, if the same object is on both the "add" and "delete" lists, then both entries are simply removed from the lists. Once the optimizations are complete, then a method is called for each `Archivable` object on every list:

- For the "add" list, the `createRecord` method is called
- For the "update" list, the `updateRecord` method is called
- For the "delete" list, the `deleteRecord` method is called

Once all of these method calls have completed, then the `CommitPoint` commits the transaction to the database.

A `CommitPoint` also provides a method to add `Restorable` objects to a restore list. The restore list is a list of objects whose contents are to be restored from the database if the `CommitPoint`'s `noCommit` method is called. In this case, the in-memory objects are refreshed to reflect what is in currently in the database.

The `noCommit` method clears all of the add, update and delete lists without performing any of the JDBC calls. This is roughly analogous to a "rollback," although in this case, the database never gets involved. For more detail, see `com.ibm.etill.framework.archive.CommitPoint`.

Cassette-specific financial objects (for example, `CassetteOrder`, `CassetteBatch`) should **never** be added directly to any of the `CommitPoint`'s lists. Only the framework financial objects (for example, `Order`, `Batch`) should be added to these lists. When the `CommitPoint` eventually calls the framework objects' `createRecord`, `updateRecord`, or `deleteRecord` methods, those methods will subsequently call that method on the corresponding cassette object's.

- *Binary Fields* are arbitrary lengths (potentially very large) of data that a cassette must record as part of one of its records. Because different database products have different characteristics and behaviors where it comes to large binary fields, the framework provides methods that will break the field up into as many segments as required by the underlying database product. The framework will also handle any other idiosyncrasies that exist. The cassette writer only has to treat the data as a single binary field. Here's how it works:
 - All binary fields are stored in a framework-supplied table named `ETBINARYDATA`.
 - Cassettes never access `ETBINARYDATA` directly. Rather, all access to the data in this table is made through the framework's access methods on the `ETillArchive` class. These are `ETillArchive.createBinaryField`,

ETillArchive.readBinaryField and ETillArchive.deleteBinaryField. Note that no "updateBinaryField" method exists. To update an existing binary field, you must create a new one and delete the old one.

- All binary fields are uniquely identified with a "uniqueKey" that is generated by the Supervisor.getUniqueKey method.
- Since the binary fields reside in a dedicated database table, references to these fields from other tables must be made using the uniqueKey. Note that ETillArchive.addExternalField is a convenience method that will allocate the uniqueKey and perform the association as a single operation.
- The ETillArchive class provides several other methods that:
 - Begin and complete transactions through JDBC
 - Simplify the assembly of prepared statements within Archivable methods
 - Build and execute SQL query commands through JDBC
 - Provide other useful functions related to the WebSphere Commerce Payments database.

For more detail, see ETillArchive.

All access to JDBC through ETillArchive must be synchronized using the ETillArchive.getArchiveLock method. This gross level lock protects against a variety of concurrency problems that have existed in JDBC drivers for different database products. In addition, the cassette must ensure that no other locks are obtained while the database lock is held. This means that the cassette must not call framework services that perform other synchronization calls when the archive lock is held. Doing so risks a deadlock situation.

Note: Gross level locking is necessary due to deadlock encountered with some JDBC drivers.

The following example shows how to add, update, and delete objects from a database:

```
//-----  
// Code example adding, updating and deleting objects from a database.  
//-----  
  
//When a new batch is opened, it is added to the database  
Supervisor.getThreadCommitPoint().addToCreateList(fwkBatch);  
  
// And when the cassette is sure it wants to keep this change  
// a commit is called  
Supervisor.getThreadCommitPoint().commit();  
  
...  
  
// An update is needed if the batch changes in some  
// significant way, for example its state.  
Supervisor.getThreadCommitPoint().addToUpdateList(fwkBatch);  
Supervisor.getThreadCommitPoint().commit();  
  
...  
  
// If deleting a batch is allowed by your protocol, a  
// cassette writer would want to remove the batch from  
// the Database.  
Supervisor.getThreadCommitPoint().addToDeleteList(fwkBatch);  
  
// If the cassette goes down an error path and decides  
// to erase the last DB operations since the last commit
```

```
// a noCommit is called and the objects on the restoreList
// are restored from the database.
Supervisor.getThreadCommitPoint().noCommit();
```

Event notification

Events are external asynchronous notifications of a change that has occurred within WebSphere Commerce Payments or one of its cassettes. When they are generated, the framework's Event Notification service sends the event to each event listener application that has registered for this type of event. See the *WebSphere Commerce Payments Programming Guide and Reference* for a more complete description of the framework events, Event Notification service, and the Event Listener.

Most cassettes will never need to take any explicit action with respect to events since the most common types of events (state change events and network management events) are generated automatically by the framework. However, it is possible for cassettes to define and generate their own events using these steps:

1. Instantiate and initialize an appropriate `com.ibm.etill.framework.eventmgr.CassetteEvent` object,
2. Add any cassette-specific parameters to the event object using the `setCassetteData` method
3. Add the new event object to the thread's `CommitPoint` through that object's `addEventToList` method
4. Call (eventually) the `CommitPoint` object's `commit` method.

When the `commit` call executes, the `CommitPoint` passes all of the events on its event list to the framework's Event Notification service for delivery to the appropriate event listeners.

Here is an example of the code to perform these steps:

```
// create cassetteEvent object
CassetteEvent ce = new CassetteEvent(this, cassetteName, merchantNumber);
// Create a hashtable and load it with our event data
Hashtable cassetteHashtable = new Hashtable();
cassetteHashtable.put("csKeyword1", "csValue1");
cassetteHashtable.put("csKeyword2", "csValue2");
// Reference hashtable from the event object
ce.setCassetteData(cassetteHashtable);
...
// Commit this object to the database and send our event to registered
// event listeners
Supervisor.getThreadCommitPoint().commit();
```

For more detail, see `CassetteEvent`.

Map your cassette AVS codes to the WebSphere Commerce Payments common AVS result codes

One of the WebSphere Commerce Payments merchant integration goals is that the installation of new cassettes into a merchant environment involves minimal work for the merchant. One framework service that supports this goal is the definition of common Address Verification Service (AVS) result codes. AVS is a primarily US-specific credit card fraud detection mechanism that may be implemented by a cassette if the payment protocol supports it. Since AVS result codes are not standardized, the framework has designed a common set of AVS result codes that all cassettes utilizing AVS should use. Thus, the merchant software will only have to understand one set of AVS codes.

If you have an existing cassette or are writing a new one, you should determine how to map your cassette-specific AVS codes to the common AVS results codes. The table below shows the framework defined common AVS result codes. The constants are defined in `com.ibm.etill.framework.payapi.PaymentAPIConstants`

Table 4. Framework defined AVS result codes

Common AVS code	PM constant name	Explanations
0	AVS_COMPLETE_MATCH	Both the 5-digit and 9-digit postal code and street address are exact matches.
1	AVS_STREETADDRESS_MATCH	The street address matches but the postal code does not match.
2	AVS_POSTALCODE_MATCH	The 5-digit or 9-digit postal code matches but the street address does not match.
3	AVS_NO_MATCH	Neither the street address nor the postal code matches.
4	AVS_OTHER_RESPONSE	This constant maps the following: address information unavailable, system unavailable maybe due to timeout, card type not supported, or transaction ineligible AVS return codes.

The following table contains suggested text messages that correspond to the AVS result code constants. To provide consistency across all cassettes, you are encouraged to use these messages when displaying the common AVS code in the UI. These message strings are not supplied by the framework, and are the responsibility of the cassette developer to supply in the appropriate properties file.

Table 5. Suggested text messages

Constant	Message text
AVS_COMPLETE_MATCH	The street address and postal code completely match.
AVS_STREETADDRESS_MATCH	The street address matches but the postal code does not.
AVS_POSTALCODE_MATCH	The postal code matches but the street address does not.
AVS_NO_MATCH	No match in street address and postal code.
AVS_OTHER_RESPONSE	Request cannot be processed at this time due to one of the following: address information is unavailable, system unavailable, maybe due to timeout, card type not supported, data not available.

Asynchronous Auto Approve

Prior to WebSphere Payment Manager Version 2.2, when the Merchant Server software issued an AcceptPayment command or a ReceivePayment command with the "autoApprove" flag (APPROVEFLAG=1), the cassette was expected to perform approval processing before completing the request and returning the APIResponse. The APPROVEFLAG keyword on the AcceptPayment and ReceivePayment API commands now allows the specification of asynchronous approvals. The APPROVEFLAG keyword can be specified as 0 (indicating that automatic approval should not occur), 1 (indicating that automatic approval should occur, or 2 (indicating that an automatic approval should occur asynchronously).

Unlike their synchronous counterpart, asynchronous automatic approvals are implemented completely within the framework. Therefore, when asynchronous approval is specified in the APPROVEFLAG, the cassette should treat the order creation step (the AcceptPayment or ReceivePayment) as if no automatic approval processing were specified at all; that is, the APIResponse should be returned once the order creation process has completed. The framework will then schedule the Approve to occur on a separate thread, thus, the approval process is initiated asynchronously. One of the benefits of doing the approval asynchronously, is that the buyer (or some other agent) does not have to wait for the approval to occur before receiving a response from the original purchase request (that is, the AcceptPayment or ReceivePayment command that was issued on behalf of the buyer). It should be noted that if autoDeposit is also specified (through the DEPOSITFLAG), the deposit is done in a synchronous manner after the approval; that is, there is no concept of asynchronous autoDeposit.

Because asynchronous automatic approvals are implemented in the framework, no cassette changes should typically be required to support this feature. From a cassette perspective, the asynchronous approval initiated by the framework will appear to the cassette as an ApprovalRequest, and should be serviced in the same way an ApprovalRequest is when the Approve API command is issued. The cassette can determine if it should perform a synchronous automatic approval in the following ways:

- For the AcceptPayment command, if the method `com.ibm.etill.framework.cassette.AcceptPaymentRequest.getApproveFlag()` returns true. Note that this method will return true ONLY if synchronous automatic approval was requested. For asynchronous autoApprove, this method returns false.
- For the ReceivePayment command, the method `com.ibm.etill.framework.payapi.Order.getApproveFlag()` returns 1. Note that this method will return the effective numeric value of the APPROVEFLAG value (0, 1 or 2) in all cases.

The following table illustrates what action should be taken by the cassette on the **AcceptPayment** and **ReceivePayment** API commands:

APPROVEFLAG	DEPOSITFLAG	Cassette action
0	0	order creation process.
0	1	order creation process. Value of 1 for DEPOSITFLAG should be ignored.
1	0	perform approval processing after order creation.
1	1	perform approval processing and deposit processing after order creation.
2	0	order creation process
2	1	order creation process

For both asynchronous and synchronous auto-approves, the following conventions are used:

- Approval amount is the amount of purchase if PAYMENTAMOUNT is not specified in the API command.
- Payment number is 1 if PAYMENTNUMBER is not specified in the API command.

- If auto-deposit is specified, the deposit amount is the same as the approval amount.

Account settings related to AcceptPayment and ReceivePayment

The AcceptPayment and ReceivePayment API commands allow the specification of autoApprove and autoDeposit flags that indicate whether approvals and deposits should be attempted automatically. In Payment Manager versions 2.1.5.0 or higher, a new feature for specifying autoApprove and autoDeposit is introduced. This feature provides merchants with an option, based on their relationship with their acquirers, to specify autoApprove or autoApprove with autoDeposit on an account basis rather than on an API basis. For example, if a particular acquirer only supports Sales transactions (autoApprove with autoDeposit), then the account settings associated with that acquirer can be specified accordingly. This allows the merchant server software to not have to pass the autoApprove and autoDeposit flags (APPROVEFLAG and DEPOSITFLAG) on each AcceptPayment/ReceivePayment command; the account settings will determine the values for those flags based on what an acquirer requires.

Since an Account represents a relationship between a merchant and an acquirer, the flags that indicate whether a purchase should be automatically approved and, optionally, deposited, are associated with the `com.ibm.etill.framework.admin.AccountAdmin` object. As a result, during creation and modification of an Account (via the API commands `CreateAccount` and `ModifyAccount`), these flags can be specified. The new framework keywords for the `CreateAccount` and `ModifyAccount` API commands are:

- `APAPPROVEFLAG` - autoApprove flag for AcceptPayment command
- `APDEPOSITFLAG` - autoDeposit flag for AcceptPayment command
- `RPAPPROVEFLAG` - autoApprove flag for ReceivePayment command
- `RPDEPOSITFLAG` - autoDeposit flag for ReceivePayment command

The `APAPPROVEFLAG` and `RPAPPROVEFLAG` can have the following values:

- 0 - no auto-approval (default)
- 1 - synchronous auto-approval
- 2 - asynchronous auto-approval (see the section “Asynchronous Auto Approve” on page 107 for details)

Since both the AcceptPayment and ReceivePayment API commands and the Account object can indicate auto-approval and autoDeposit, the precedence rules are as follows:

- If the value of the API flag is not 0, use the value from the API.
- If the value of the API flag is 0 or not specified, use the value in the account object.

For example, for an AcceptPayment command, if the `APPROVEFLAG` is 0, and the corresponding account object’s `APAPPROVEFLAG` is 1, then a synchronous autoApprove should occur.

Cassette changes required for AcceptPayment

If your cassette only supports AcceptPayment, then the code you write to handle autoApprove and autoDeposit from the command line will automatically handle the account-based flags. This is due to the fact that the framework ensures that the autoApprove and autoDeposit flags in the `AcceptPaymentRequest` contain the appropriate value based on the precedence rules stated above. The framework can achieve this because of the following sequence of events:

- When the AcceptPayment command comes in, the framework creates a framework Order object (com.ibm.etill.payapi.Order), and then calls the cassette method newCassetteOrder().
- The cassette creates a cassette-specific order object. It is the cassette's responsibility at this point to associate an account with the newly created cassette order.
- The framework obtains the account number from the cassette order and retrieves the associated AccountAdmin object so that the account settings for the APAPPROVEFLAG and APDEPOSITFLAG can be determined.
- The framework then asks the framework Order object to adjust its values for the approve flag and deposit flag based on and the precedence rules. The method used to accomplish this is

```
com.ibm.etill.framework.payapi.Order.adjustAutoFlagsBasedOnAccountSettings( AccountAdmin, boolean)
```

The AccountAdmin object obtained in the previous step is the first parameter to this method. The boolean parameter is an indication if the command being processed is an AcceptPayment command.

- The framework creates an AcceptPaymentRequest, populating it with the newly determined values for autoApprove and autoDeposit. The request is sent to the cassette via the service() method.
- As always, the cassette can ascertain whether an autoApprove or autoApprove with autoDeposit should be performed by calling the following methods:

```
com.ibm.etill.framework.cassette.AcceptPaymentRequest.getApproveFlag()
com.ibm.etill.framework.cassette.AcceptPaymentRequest.getDepositFlag()
```

Cassette changes required for ReceivePayment

If your cassette supports ReceivePayment, then cassette-specific code is required since the cassette cannot determine which account is associated with the order until all protocol-specific events have occurred (i.e., until the cassette has received all of the payment instructions from the consumer). Once the cassette has determined which account is associated with the order, it must do the following:

- Obtain a reference to the framework's account object (com.ibm.etill.framework.admin.AccountAdmin) that is associated with the order.

- Call the following:

```
com.ibm.etill.framework.payapi.Order.adjustAutoFlagsBasedOnAccountSettings( AccountAdmin, boolean )
```

passing in the Account object that was obtained in the previous step, and false (which indicates that this is not an AcceptPaymentRequest). This causes the framework Order object to adjust its values for the approve flag and deposit flag based on and the precedence rules.

- From this point forward, the cassette should determine if an autoApprove or autoApprove with autoDeposit should be performed by calling the following methods:

```
com.ibm.etill.framework.payapi.Order.getApproveFlag()
com.ibm.etill.framework.payapi.Order.getDepositFlag()
```

Telling the framework which order creation commands are supported by the cassette

The WebSphere Commerce Payments user interface allows the setting of the account-based autoApprove and autoDeposit flags in the Advanced Settings section of the Account Settings window. By default, options are displayed for both the AcceptPayment API command and the ReceivePayment API command. If your cassette only supports one of these commands, then you can ensure that these

options will be displayed only for the supported commands in the user interface by overriding the following methods and returning true or false, as appropriate:

```
framework.cassette.Cassette.isAcceptPaymentSupported()  
framework.cassette.Cassette.isReceivePaymentSupported()
```

Configurable approval expiration

Configurable approval expiration provides a cost-saving option to merchants. It is not uncommon for an order to be delayed enough such that the original approval authorization expires. In this case, the merchant may incur charges when trying to deposit the funds. Configurable approval expiration support allows the merchant application to recognize this situation and avoid it entirely.

ApprovalExpiration parameter

An `ApprovalExpiration` parameter is provided with the `CreateAccount` and `ModifyAccount` commands to specify the period of time after which a given approval should be considered expired. A payment in the `Approved` state will enter the `ApprovalExpired` state after the specified period of time has elapsed. A merchant application can use the `ApproveReversal` command either to put a Payment back in the `Approved` state or to void it. For more merchant application details, see the *WebSphere Commerce Payments Programming Guide and Reference*. Each cassette is responsible for specifying whether or not it supports approval expiration.

Cassette enablement

The WebSphere Commerce Payments framework assumes that the cassette does not support approval expiration. To enable approval expiration, your cassette must do the following:

- Indicate that the cassette supports approval expiration. This is done by overriding `com.ibm.etill.framework.cassette.Cassette.isApprovalExpirationSupported()` and returning true.
- Handle payments that have expired. The cassette must process the `ApprovalExpirationRequest` sent by the framework. Like all requests sent by the framework, the `ApprovalExpirationRequest` is sent to the cassette through the `service()` method. The cassette should do whatever is required to put the payment in `PAYMENT_APPROVALEXPIRED` state.
- Allow an `ApproveReversal` when the payment is in the `PAYMENT_APPROVALEXPIRED` state.
- Put the payment in `PAYMENT_APPROVALEXPIRED` state if a deposit fails because it expired.

When a cassette supports approval expiration, the framework does the following:

- Enables the `Approval Expiration` field on the `Account Advanced Settings` screen in the WebSphere Commerce Payments user interface
- Manages the framework account approval expiration value
- Manages the expiration time of the payment:
 - If the framework finds that the value has been filled in by the cassette, it will use the cassette's value rather than compute one by adding the account's expiration delta to the payment's authorization time. This allows cassettes to provide the real expiration time.
 - If a payment expires, the framework will send an `ApprovalExpirationRequest` to the cassette.

Purchasing card support

Purchasing cards (also known as procurement cards) are credit cards that a business can offer its departments or employees to allow them to buy business related items. Typically, a business will make arrangements with the card issuer to govern the purchases that cardholders can make. For example, maximum limits can be imposed and the cards can be restricted to allow purchases of certain items only (for example, only stationery goods). Purchasing cards can also have pre-programmed limits for purchase amounts. Purchase-related details (such as the tax amount, and merchant category code) and the details of the items being ordered through a purchasing card are passed to the financial network so that the authorization of the purchase can be influenced by the details of the goods being ordered. Purchasing cards are a form of payment commonly used by many businesses because it streamlines the corporate purchasing process.

Three levels of purchasing card information are supported in WebSphere Commerce Payments and are generally known in the industry as Level I, II, and III data:

- Level I data includes standard commercial transaction data for the purchase which may include the total purchase amount, the date of purchase, commodity code, the merchant's name, and other data elements as defined by the credit card associations or similar entity.
- Level II data adds additional data to Level I data about each purchase, including the merchant category code, sales tax amount, and other data.
- Level III data includes full line-item detail in addition to the data in Level II which includes unit cost, quantities, unit of measure, product codes, product descriptions and other data elements.

Table 6. lists the purchasing card data entities defined by the WebSphere Commerce Payments framework that are commonly used. If your cassette needs to support additional purchasing card data, you can define additional data entities for your use.

Table 6. Purchasing card parameters

Purchasing card parameter	Description
\$PCARD.SHIPPINGAMOUNT	Total shipping/freight amount for the order.
\$PCARD.DUTYAMOUNT	Total amount of duties or tariff for the order.
\$PCARD.DUTYREFERENCE	Reference number assigned to the duties or tariff for the order.
\$PCARD.NATIONALTAXAMOUNT	Total amount of national tax (sales or VAT) applied to the order.
\$PCARD.NATIONALTAXRATE	National tax (sales or VAT) rate applied to the order.
\$PCARD.LOCALTAXAMOUNT	Total amount of local tax applied to the order.
\$PCARD.OTHERTAXAMOUNT	Total amount of other taxes applied to the order.
\$PCARD.TOTALTAXAMOUNT	Total amount of all taxes applied to the order.
\$PCARD.MERCHANTTAXID	Tax identification number of the merchant.
\$PCARD.ALTERNATETAXID	Alternate tax ID number of the merchant.
\$PCARD.TAXEXEMPTINDICATOR	The tax exempt indicator for the order.
\$PCARD.MERCHANTDUTYTARIFFREFERENCE	Duty or tariff reference number assigned to the merchant.
\$PCARD.CUSTOMERDUTYTARIFFREFERENCE	Duty or tariff reference number assigned to the cardholder.
\$PCARD.SUMMARYCOMMODITYCODE	Commodity code that applies to the entire order.
\$PCARD.MERCHANTTYPE	Type of merchant.
\$PCARD.MERCHANTCOUNTRYCODE	The ISO country code portion of the merchant's location.

Table 6. Purchasing card parameters (continued)

Purchasing card parameter	Description
\$PCARD.MERCHANTCITYCODE	City name portion of the merchant's location.
\$PCARD.MERCHANTSTATEPROVINCE	Name or abbreviation of the state or province of the merchant's location.
\$PCARD.MERCHANTPOSTALCODE	Postal code of the merchant's location.
\$PCARD.MERCHANTLOCATIONID	Identifier that the merchant uses to specify one of its locations.
\$PCARD.MERCHANTNAME	Name of the merchant.
\$PCARD.SHIPFROMCOUNTRYCODE	The ISO country code portion of the location where the goods are shipped from.
\$PCARD.SHIPFROMCITYCODE	City name portion of the location where the goods are shipped from.
\$PCARD.SHIPFROMSTATEPROVINCE	Name or abbreviation of the state or province of the location where the goods are shipped from.
\$PCARD.SHIPFROMPOSTALCODE	Postal code of the location where the goods are shipped from.
\$PCARD.SHIPFROMLOCATIONID	An identifier that the merchant uses to specify one of its locations where the goods are shipped from.
\$PCARD.SHIPTOCOUNTRYCODE	The ISO country code portion of the location where the goods are shipped to.
\$PCARD.SHIPTOCITYCODE	City name portion of the location where the goods are shipped to.
\$PCARD.SHIPTOSTATEPROVINCE	Name or abbreviation of the state or province of the location where goods are shipped to.
\$PCARD.SHIPTOPOSTALCODE	Postal code of the location where the goods are shipped to.
\$PCARD.SHIPTOLOCATIONID	An identifier that the merchant uses to specify the location where the goods are shipped to.
\$PCARD.MERCHANTORDERNUMBER	Merchant order number.
\$PCARD.CUSTOMERREFERENCENUMBER	Reference number assigned to the order by the cardholder.
\$PCARD.ORDERSUMMARY	Summary description of the order.
\$PCARD.CUSTOMERSERVICEPHONE	Merchant's customer service telephone number.
\$PCARD.DISCOUNTAMOUNT	The discount amount applied to the order.
\$PCARD.SHIPPINGNATIONALTAXRATE	The national (sales or VAT) tax rate applied to the shipping amount.
\$PCARD.SHIPPINGNATIONALTAXAMOUNT	The national (sales or VAT) tax applied to the shipping amount.
\$PCARD.NATIONALTAXINVOICEREFERENCE	The national (sales or VAT) tax invoice reference number for the order.
\$PCARD.PRINTCUSTOMERSERVICEPHONENUMBER	Specifies if the issuer may print the merchant's customer service phone number on the cardholder's statement.
<i>Line item data</i>	
\$ITEM.COMMODITYCODE	Commodity code for the line item.
\$ITEM.PRODUCTCODE	Product code for the line item.
\$ITEM.DESRIPTOR	A description of the line item.
\$ITEM.QUANTITY	The quantity for the line item.
\$ITEM.SKU	The stock keeping unit (SKU) of the line item.
\$ITEM.UNITCOST	Unit cost of the line item.
\$ITEM.UNITOFMEASURE	Unit of measure for the line item.
\$ITEM.NETCOST	Net cost per unit of the line item.
\$ITEM.DISCOUNTAMOUNT	Amount of discount applied to the line item.
\$ITEM.DISCOUNTINDICATOR	Required only if any other line item information is present. Indicates if a discount was applied.
\$ITEM.NATIONALTAXAMOUNT	Amount of national tax (sales or VAT) applied to the line item.
\$ITEM.NATIONALTAXRATE	National tax (sales or VAT) rate applied to the line item.

Table 6. Purchasing card parameters (continued)

Purchasing card parameter	Description
\$ITEM.NATIONALTAXTYPE	Type of national tax applied to the line item.
\$ITEM.LOCALTAXAMOUNT	Amount of local tax applied to the line item.
\$ITEM.LOCALTAXRATE	Local tax rate applied to the line item.
\$ITEM.OTHERTAXAMOUNT	Amount of other taxes applied to the line item.
\$ITEM.TOTALCOST	The total cost of the line item.

Framework and cashier support of purchasing cards

To support the passing of purchase-related details to a financial network, the WebSphere Commerce Payments framework does the following:

- It defines the protocol data that represents the purchasing card information, and associated secondary return codes. For example:
 - \$PCARD.SHIPPINGAMOUNT – This parameter represents the total shipping/freight amount for the order.
RC_CASSETTE_PCARD_SHIPPING_AMOUNT – The associated secondary return code for this parameter.
 - \$ITEM.QUANTITY – This parameter specifies the quantity for the line item.
RC_CASSETTE_ITEM_QUANTITY – The associated secondary return code for this line item parameter.

Table 6 on page 112 lists the protocol data you should use for consistency. If, however, your cassette has special requirements and you must process other types of data beyond that which is provided, you can add your own.

- It provides the following class to help your cassette manipulate the purchasing card data passed in on an API command. For more information about this class, refer to the Javadoc:
 - com.ibm.etill.framework.cassette.PurchaseCardData
- It allows repeating protocol data to be passed in on API commands through the ".n" convention (for example, \$ITEM.QUANTITY.3) so that the existing cassette parameter validation still works. The framework command processing strips the ".n" portion of the protocol data keyword before passing it to the cassette. The cassette then does parameter validation on \$ITEM.QUANTITY.

To support the line item detail of purchasing card data, the WebSphere Commerce Payments cashier supports the possibility of multiple values for a single parameter. For example, suppose that one of the line item protocol data is \$ITEM.QUANTITY, and a purchase is being made that includes four line items:

```
$ITEM.QUANTITY.1
$ITEM.QUANTITY.2
$ITEM.QUANTITY.3
$ITEM.QUANTITY.4
```

The cashier allows multiple rows to be returned when using a database parameter. (Prior to version 3.1, only one row could be returned when using a database parameter.) In the previous example, the cashier can issue a database query, and for the \$ITEM.QUANTITY database parameter, generate the protocol data for each of the four rows returned from the database query.

To illustrate the concept of allowing multiple rows to be returned, note the following example. The Cashier allows multiple rows to be returned for the \$ITEM.QUANTITY database parameter:

```

<SelectStatement id="5" allowMultiples="true">
  SELECT * FROM ... WHERE ...

<Parameter name="$ITEM.QUANTITY">
  <DatabaseValue statementID="5" columnName="ITEMQTY"/>
</Parameter>

```

The cashier also supports the issuing of the Deposit command in the event a cassette needs to pass purchasing card data during a deposit.

Guidelines when passing data in on commands

As a cassette developer, you need to decide the appropriate WebSphere Commerce Payments commands that will support the passing of purchasing card data. Here are some guidelines:

- If the payment processor you are interfacing with accepts purchasing card data at authorization, then consider allowing purchasing card data to be sent on the AcceptPayment and Approve commands.
- If the payment processor you are interfacing with accepts purchasing card data at the settlement phase, then consider allowing purchasing card data to be sent on the AcceptPayment, Approve, and/or Deposit commands.

You must also decide what financial object should be associated with the purchasing card data. Obvious logical choices are the Order object and/or the Payment object. If you decide to use the Order object, be careful about associating multiple payments with the order. It usually does not make sense to associate the same set of purchasing card data (especially line item detail) with two different payments.

Error logging

WebSphere Commerce Payments provides cassette developers with the ability to generate informational and error messages that can easily be translated to provide support for foreign languages. The `com.ibm.etill.framework.log.ErrorLog` class provides a set of methods that retrieve the cassette's messages from its properties file (described below), includes any substitutions, and logs them to the `activity.log` file for the Payments instance (accessible through the WebSphere Application Server logs directory). This error logging facility is available within the Payment Servlet. For details, see `com.ibm.etill.framework.log.ErrorLog`.

Each cassette must provide its own properties file that defines all the error messages that the cassette can produce. The directory that contains this file must be specified in the CLASSPATH when starting WebSphere Commerce Payments.

The naming convention for the properties file is:

```
cassetteName.properties
```

where `cassetteName` is the name of your cassette as defined in the PAYMENTSYSTEMNAME field of the ETCASSETTECFG database table.

The use of this file simplifies the process of internationalizing the code since the messages are retrieved by a key (the message number). Different versions of the `.properties` file can be provided for different languages.

Each message should be defined in the properties file in the following format:

```
<messageNumber> = <messageText>
```

where:

messageNumber

is an alphanumeric identification of the error message that is unique within your cassette. When the error message is displayed, WebSphere Commerce Payments constructs a new message identifier as follows:

```
CEP<cassetteName><messageNumber>
```

messageText

is the error description that can optionally contain parameters provided by your cassette. WebSphere Commerce Payments uses the Java MessageFormat format to parse the message text.

You can also use the .properties file to contain translatable phrases or strings that may later be used as message substitution values. For example:

```
text_welcome = Willkommen
```

The text from such an entry can be retrieved using the `ErrorLog.getMessageText` method. Note that this example also shows that the keys in the properties file do not have to be numeric. While this practice is acceptable for short phrases and keywords as illustrated, you should follow the convention of using the numeric message number values as the keys for your informational and error messages.

This example shows how to use the `ErrorLog` object to write error messages to the error log. In this example, the error is a database access error.

```
//-----
// Code example of using the ErrorLog object
//-----

...

catch(SQLException e)
{
    // Example of a logError() call without any substitutions
    ErrorLog.logError("ExampleCassette", "0205", e);

    // Example of a logError() call with two substitutions
    ErrorLog.logError("ExampleCassette", "001122", e,
        "Substitute Text 1", "Substitute Text 2");
}

...

//-----
// ExampleCassette.properties file entry example
//-----

0205 = An SQL exception occurred while accessing the database.

001122 = An SQL exception occurred.
        Substitution text1: {0} and Substitution text2: {1}
```

Note: Currently WebSphere Commerce does not support the viewing of third-party cassette messages in the WebSphere Commerce symptom database. The symptom database is described in the *WebSphere Commerce Administration Guide*.

Return code messages

One of the WebSphere Commerce Payments merchant integration goals is that the installation of new cassettes into a merchant environment involves minimal work for the merchant. One requirement to support this goal is that merchant software should not have to translate each WebSphere Commerce Payments return code pair

into a message that can be displayed to the buyer. Thus, in addition to the Error Logging support described above, WebSphere Commerce Payments provides cassette developers with the ability to add textual descriptions to the return codes associated with the AcceptPayment and ReceivePayment API commands.

Every cassette will have its own collection of protocol data that is needed on the AcceptPayment and/or ReceivePayment API commands. Some of this data may be entered by the buyer on the checkout/buy page, for example, credit card number. If the buyer enters data that fails WebSphere Commerce Payments validation, the API response will contain non-zero primary and secondary return codes indicating the error. In a number of cases, the merchant software will want to display a message directly to the buyer. For example, "The credit card number is not valid". As new cassettes are employed, it is unreasonable to expect the merchant software to have to map each potential return code pair into a message that can be displayed to the buyer.

Both the framework and the cassette can return non-zero primary/secondary return code pairs that are generated as a result of the AcceptPayment and ReceivePayment API commands. Each of these return code pairs that are generated by the framework are mapped to a message that resides in the same framework properties file that is used for Error Logging. That message will be returned along with the primary and secondary return codes in the PSAPIResult document. Two new optional attributes to the PSAPIResult XML element have been added: "buyerMessage", which contains a message appropriate to be displayed to the shopper and "merchantMessage" which contains a message for the merchant to resolve. It is important to note that either a buyerMessage or merchantMessage can be returned, but not both. In order to allow the merchant server software to determine which messages should be displayed to the buyer, and which messages it needs to resolve itself, the message number will be flagged with a "B" for buyer or an "M" for merchant to indicate which party should receive the message.

Just as the framework does, cassette developers should also provide messages associated with return code pairs that can be returned on AcceptPayment and ReceivePayment. These messages should be added to the cassette's properties file. Return codes that are generated as a result of autoApprove and autoDeposit processing of AcceptPayment and ReceivePayment should also result in return code messages.

Payment Servlet processing

For each non-zero return code generated during the AcceptPayment or ReceivePayment API commands, the Payment Servlet attempts to retrieve the associated message text as follows:

- Format the primary and secondary return code into the message number. For example, if the primary return code is 3 and the secondary return code is 13, then the message number would be PRC3SRC13.
- Look in the cassette properties file for a buyer message with a number of PRC3SRC13B. If found, return the text as a buyerMessage in the PSAPIResult document. If not found, look for a merchant message with a number of PRC3SRC13M. If found, return the text as a merchantMessage in the PSAPIResult document.
- If the message number is not found in the cassette properties file, look in the framework properties file for the buyer message (PRC3SRC13B). If found, return the text as a buyerMessage. If not found, look for the merchant message (PRC3SRC13M). If found, return it as a merchantMessage.

- If the text was not found, the `buyerMessage` and `merchantMessage` fields are not returned in the `PSAPIResult` document.

Return code message syntax:

Each `messageNumber` that is associated with a return code message should have the following format in the cassette's properties file:

```
PRCxxxxSRCyyyy[B|M]
```

where "xxxx" is the primary return code and "yyyy" is the secondary return code. The suffix of "B" or "M" indicates who should receive the message. The "B" indicates buyer and the "M" indicates the merchant. The xxxx and yyyy values are *not* padded with zeroes. For example, a message with a primary return code of 3 and a secondary return code of 13 destined for the buyer would be coded as `PRC3SRC13B`.

Example

Using the previous example from the Error Logging section, the cassette unique return code messages are added at the end of the properties file. Let's assume that our sample cassette requires a credit card number, expiration date, order description, and success URL to be passed into the `AcceptPayment` command. Also, let's assume that the credit card number and expiration date are entered by the buyer, and the order description and success URL are provided by the merchant. The cassette properties file would contain:

```
//-----
// ExampleCassette.properties file entry example
//-----
0205 = An SQL exception occurred while accessing the database.
001122 = An SQL exception occurred.
.
.
.
PRC3SRC1015B = A credit card number was not specified.
PRC3SRC1016B = The expiration date was not specified.
PRC3SRC10001M = The order description was not specified.
PRC3SRC10002M = The Success URL was not specified.
```

The cassette developer only needs to add entries for those error messages not defined in the framework's properties file (`PMFramework.properties`). The framework's properties file contains some messages for all of the common protocol data parameters that are defined in `com.ibm.etill.framework.payapi.PaymentAPIConstants`. The cassette developer must add return code messages to the cassette's properties file for cassette-specific protocol data that is not defined in `com.ibm.etill.framework.payapi.PaymentAPIConstants` and for common protocol data parameters that do not have the necessary message specified in the framework's properties file. In addition, if the cassette developer can override messages defined in the framework's properties file if it chooses, since the cassette properties file is searched first. To look at the framework's properties file to determine the return code messages that are defined, do the following:

- In the WebSphere Commerce Payments installation directory, find the file `eTillClasses.jar`:
`Payments_installdir/wc.mpf.ear/lib/eTillClasses.jar`
- Copy `eTillClasses.jar` to a directory of your choice and then extract the files from the archive via the `jar xvf eTillClasses.jar` (note that this command will expand the entire contents of the `eTillClasses.jar` into the current directory, so you should not do this in the WebSphere Commerce Payments installation directory).

- The file `PMFramework_xx.properties` (where `xx=locale`) contains the error messages defined by the framework. For example `PMFramework_en.properties` contains the english messages.

Debug tracing

Through the use of the WebSphere JRas trace service, WebSphere Commerce Payments provides a flexible trace facility for recording internal events of various types, including:

- Communication events (for example, connection established/dropped, data read/write)
- Function entry and exit
- Database read, write and `commit` (generated by framework)
- Debug (a catch-all for any type of debug tracing you want to do)
- Command received (generated by framework)
- Financial object state change (generated by framework)
- Error occurred
- Start work item (generated by framework)
- Trace information (generated by framework)
- Cassette-specific messages

For a complete description of how administrators or users can use the trace facility, see the *WebSphere Commerce Administration Guide*.

As indicated, most of these trace entries are generated within the framework, but a few are of particular interest to cassette writers:

Communication events

If your cassette implements any ComPoints with their own communication support (that is, actually sending and receiving data across some communication medium), you should generate the appropriate trace entries to ensure that a support person gains a full understanding of when and how much data was sent and received by the ComPoint.

Function entry and exit

Most of your methods should implement these trace points. Trivial methods, such as simple getters and setters, can probably bypass these.

Debug

This is a catch-all category for any type of information you find useful for debugging your cassette.

Error occurred

Captures failure information at the point where the failure occurred. This is often the first type of entry a support person will search for since it can pinpoint where a failure was first detected.

Cassette-specific messages

Four different trace events are available for cassettes to define for their own purposes. If you choose to do so, your documentation should describe how each of the cassette events are used by your cassette.

Tracing is controlled completely through the `com.ibm.etill.framework.log.Trace` class, which is available within the Payment Servlet. This class provides methods

to set and examine the current trace selectivity for the cassette and to generate each of the various types of trace entries. For more detail, see `com.ibm.etill.framework.log.Trace`.

Trace methods take strings and do not look up messages in resource bundles. There is no support for translation of trace messages.

Enabling trace

You can enable tracing through the WebSphere trace service which is available through the WebSphere Application Server administrative console:

1. From the WebSphere Application Server administrative console, click **Problem Determination > Logging and Tracing** in the console navigation tree, then click the WebSphere Commerce Payments *application server* > **Diagnostic Trace**.
2. If the server is running, select the **Runtime** tab. If the server is stopped, select the **Configuration** tab.
3. Click **Enable Trace**. In the **Trace Specification** field, enter the following trace string. (You can enter the trace string directly, or generate it using the graphical trace interface. Click **Modify** to start the graphical trace interface if desired.)

```
Payments_JRAS_component.=all=enabled
```

where `Payments_JRAS_component` is the component you want to trace:

```
com.ibm.websphere.commerce.payments.MPF
com.ibm.websphere.commerce.payments.MPFUI
com.ibm.websphere.commerce.payments.SampleCheckout
com.ibm.websphere.commerce.payments.IBM_cassetteName
com.ibm.websphere.commerce.payments.third-party_cassetteName
```

Examples of other trace specification strings follow:

```
com.ibm.websphere.commerce.payments.MPF=all=enabled
com.ibm.websphere.commerce.payments.MPFUI=all=enabled
com.ibm.websphere.commerce.payments.MPF=entryExit=enabled
com.ibm.websphere.commerce.payments.MPF=debug=enabled
com.ibm.websphere.commerce.payments.*=all=enabled
com.ibm.websphere.commerce.payments.my_Cassette=all=enabled,event=disabled
```

In the fifth example (`com.ibm.websphere.commerce.payments.*=all=enabled`), tracing is enabled for all component names starting with `com.ibm.websphere.commerce.payments`.

In the last example, trace is enabled for a third-party cassette component, and then event tracing is disabled for that component.

Debug tracing is considered JRas trace level 3. Table 8 on page 122 shows the JRas trace levels for WebSphere Commerce Payments trace types.

For more information about tracing WebSphere Commerce Payments JRas components, refer to the *WebSphere Commerce Administration Guide*. For more information about using the trace facility or the grammar involved in entering trace specification strings, refer to the "Enabling trace" information in the WebSphere Application Server Info Center.

Trace output

You can control which file the trace text is written in WebSphere Application Server by specifying the appropriate trace output information on the Configuration tab of the WebSphere trace service. (Prior to WebSphere Commerce Payments version 5.5, the `PMTrace1.log` and `PMTrace2.log` files in the WebSphere Commerce Payments log directory were used to store trace output.) The trace output can be

written directly to an output file, or stored in memory and written to a file on demand. The trace output files can be read with a text editor and do not require any special formatting utility.

The following is an example of the trace format:

```
[02.01.14 16:26:45.025 EDT] 53ccc3c5 MPF E * ** Framework version: 5.5.0.0
```

Timestamp

Timestamp in modified ISO format, controlled by WebSphere.

Thread ID

A hexadecimal ID for the thread (for example, 53ccc3c5).

Component ID mapping

The component name (com.ibm.websphere.commerce.*); for example, MPF.

```
com.ibm.websphere.commerce.payments.MPF
com.ibm.websphere.commerce.payments.MPFUI
com.ibm.websphere.commerce.payments.SampleCheckout
com.ibm.websphere.commerce.payments.IBM_cassetteName
com.ibm.websphere.commerce.payments.third-party_cassetteName
```

EventType

A one-character field that indicates the type of the trace event. Possible values include:

- > – a trace entry of type method entry
- e – a trace entry of type event
- d – a trace entry of type debug

For more information about other possible EventType values, refer to the WebSphere Application Server InfoCenter. Table 7 lists trace types and shows how they map to JRas trace levels.

Class name*

Only collected for function entry/exit calls. For other calls, this field is blank.

Method name**

Only collected for function entry/exit calls. For other calls, this field is blank.

Text The actual trace data.

For more information about how to interpret trace output, refer to the WebSphere Application Server InfoCenter.

Table 7. Trace type descriptions

Trace type (level)	JRas trace level	Description of trace performed
all		Performs tracing of all types (event, entryExit, debug)
event	1(E)	API usage Error occurred
entryExit	2 (>)	Database access Communications Function entry/exit
debug	3(D)	Communications data Debug Realm Other


```

...

//-----
// The code below adds trace entries to the Trace file.
//-----
public void someMethod
{
    // Trace entry when entering a method.
    // This trace line will not be displayed in the trace file because
    // traceFunctionEntry was not set above.
    // To set traceFunctionEntry for tracing, the following line is needed:
    //   Trace.setTraceSettings("CASSET", Trace.FUNC_ENTRY);
    if (Trace.isAnyoneTracing())
        Trace.traceFunctionEntry("CASSET", "Example.someMethod");

    ...

    // This trace line will be displayed in the trace file
    // because traceDebug was set above.
    if (Trace.isAnyoneTracing())
        Trace.traceDebug("CASSET", "ExComp.someMethod: Example successful.");

    ...

    // Trace entry when exiting a method.
    // This trace line will not be displayed in the trace file
    // because traceFunctionExit was not set above.
    // To set traceFunctionExit for tracing, the following line is needed:
    //   Trace.setTraceSettings("CASSET", Trace.FUN_EXIT);
    if (Trace.isAnyoneTracing())
        Trace.traceFunctionExit("CASSET", "Example.someMethod");
}

```

User interface support

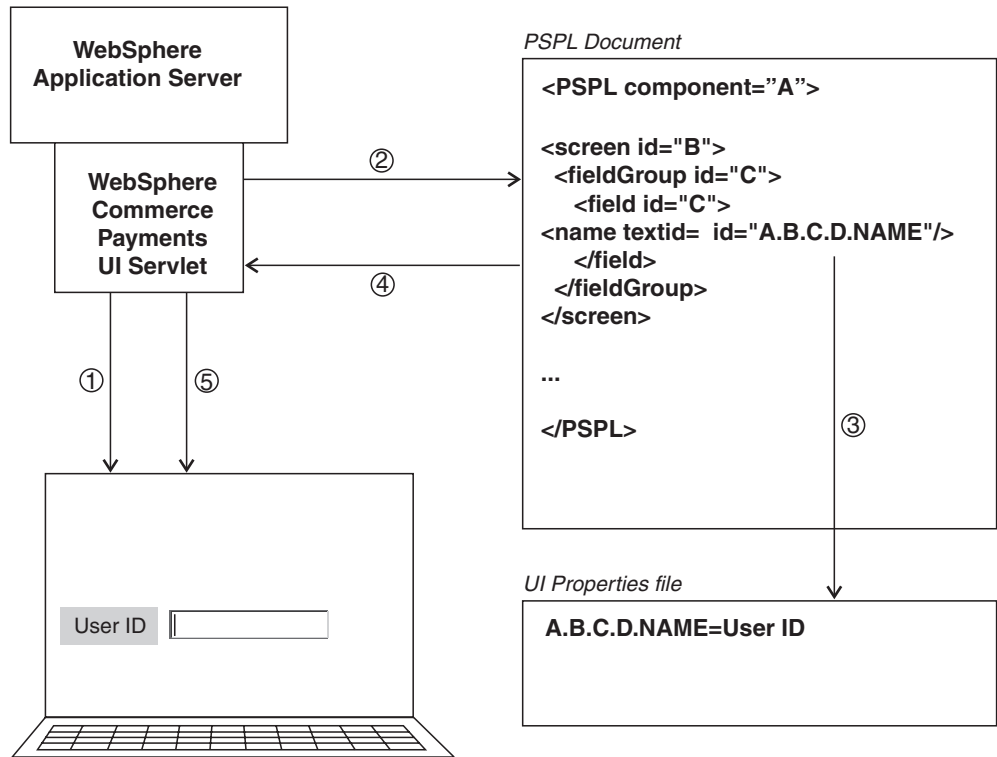
Administrators and merchants should use the WebSphere Commerce Payments User Interface Servlet to perform administrative and payment processing tasks. The user interface organizes and displays data to users through an HTML "wrapper" around WebSphere Commerce Payments HTTP requests and XML responses. The WebSphere Commerce Payments framework allows cassette writers to customize the user interface screens to display payment protocol-specific attributes. The mechanism that allows such customization is the Payment Server Presentation Language (PSPL).

Note: To display payment protocol-specific attributes in the user interface, these attributes must be defined in PSPL and externalized through the XDM, as described in the following sections of this topic.

Payment Server Presentation Language

PSPL is an XML-based language that defines the structure of HTML documents presented by the user interface. PSPL documents are divided into multiple screens, each representing a different panel in the user interface. Each user interface screen is created using specifications defined in PSPL format. The figure below illustrates how the user interface servlet uses PSPL. As shown in the figure, the user interface retrieves PSPL documents and uses these to construct an HTML document for presentation to the user. The appearance of this HTML document is defined by the

PSPL.



- ① The browser sends a request to the WebSphere Commerce Payments user interface.
- ② The user interface finds the PSPL document and requests a particular screen.
- ③ Text for the PSPL is looked up in the corresponding UI properties file.
- ④ All information to build a panel is returned to the user interface.
- ⑤ The user interface formats the PSPL information into an HTML document, and returns it to the web browser.

PSPL document overview

The structure of a PSPL document is defined in the `pspl.dtd` file located in the `pspl` directory under `WC_installdir/payments/wc.mpf.ear/payments.war/`. The user interface (UI) draws on multiple PSPL documents, both framework and cassette, when constructing an HTML document to be presented to the user. There are three framework PSPL documents: `payment.PSPL`, `admin.PSPL` and `reports.PSPL`. These framework PSPL documents determine how framework attributes are displayed in the UI, regardless of which cassettes are installed. A cassette writer must provide a PSPL document for their cassette in order to identify cassette-specific attributes to be displayed in the UI. These cassette-specific attributes can be associated with extensions to existing framework screens, or with new cassette defined screens. The following is a table of the framework's user interface files, where they are located, and what they provide. *Payments_installdir* indicates the location of the installed WebSphere Commerce Payments component (*WC_installdir/payments*)

Filename	Location	Description
<code>admin.PSPL</code>	<code>Payments_installdir/wc.mpf.ear/payments.war/pspl</code>	Defines the layout for framework Administration screens.
<code>payment.PSPL</code>	<code>Payments_installdir/wc.mpf.ear/payments.war/pspl</code>	Defines the layout for framework Payment screens.
<code>reports.PSPL</code>	<code>Payments_installdir/wc.mpf.ear/payments.war/pspl</code>	Defines the layout of the reports in the user interface.

Filename	Location	Description
PMUI.properties	<i>Payments_installdir</i> /eTillUI.jar	Contains the content used in framework admin, payments and reports PSPL (there is one of these for each supported language).
pspl.dtd	<i>Payments_installdir</i> /wc.mpf.ear/payments.war/pspl	This file is the XML definition of all PSPL files.

When a cassette that you have written is added to a Payments instance through the WebSphere Configuration Manager, its PSPL files are also added to the *WC_installdir*/payments/wc.mpf.ear/payments.war/pspl directory.

The UI servlet searches for PSPL files in the directories which are listed in the CLASSPATH environment variable. Therefore, your cassette's PSPL files must reside in a directory which is a member of the WebSphere Application Server's CLASSPATH. For recommendations on where PSPL files should be placed during cassette installation, see "Installation and uninstallation considerations and steps" on page 173.

General structure

A cassette PSPL document consists of:

- Header information such as the cassette name and the associated help file
- One or more *screens* which describe extensions and configurations to the framework screens
- Zero or more *screens* which describe new cassette-specific screens associated with MerchantCassetteObjects or SystemCassetteObjects
- Each screen in turn can contain a header, a trailer, one or more field groups, and messages
- A *field group* is a logical grouping of data and contains a header, trailer, one or more *fields*, and *messages*
- The *field* then describes the display attributes of the data to be presented to the end user
- One or more *message* elements which describe messages to be displayed.

Each of the above elements is identified by a unique identifier and associated attributes. See "PSPL reference" on page 133 for further detail about these elements.

Here is an example skeleton of a cassette PSPL file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE PSPL SYSTEM "pspl.dtd">

<!--
*****
* The PSPL element is the top-level element of all PSPL
* documents. The component attribute identifies the cassette
* to the framework, in this case "MyCassette".
*****
-->
<PSPL component="MyCassette" ...>

<!--
*****
* The screen element "MyCassetteAccount" extends the
* framework screen PSMerchantAccount. Here, the cassette
* defines it's extensions to the AccountAdmin object,
* specifically, "accountField1" and "accountField2". In
```

```

* addition, all messages associated with this screen that
* can be displayed in the UI are defined.
*****
-->
<screen id="MyCassetteAccount" extends="PSMerchantAccount">
  <fieldGroup id="MyCassetteAccountDetails" advanced="0">
    ...
    <field id="accountField1" ...>
      ...
    </field>
    <field id="accountField2" ...>
      ...
    </field>
  </fieldGroup>

  <message id="PRC=5-SRC=11008" type="error" ...>
  <message id="PRC=6-SRC=11008" type="error" ...>
  ...
</screen>

<!--
*****
* The screen element "MyCassetteBrand" configures the
* framework screen PSMerchantAccount. "configures" is used
* when the cassette has defined additional cassette-specific
* objects which are associated with framework objects. Here,
* the cassette defines a MerchantCassetteObject called "brand"
* that is associated with an account.
*****
-->
<screen id="brand" updateID="BRAND" configures="PSMerchantAccount">
  <fieldGroup id="MyCassetteBrand">
    ...
    <field id="brandID" updateID=$BRANDID ...>
      ...
    </field>
  </fieldGroup>

  <message id="PRC=5-SRC=11009" type="error" ...>
  <message id="PRC=6-SRC=11009" type="error" ...>
  ...
</screen>

<!--
*****
* The screen element "MyCassetteOrder" extends the
* framework screen PSOrder. Here, the cassette
* defines it's extensions to the Order object,
* specifically, "pan" and "expirationDate". In
* addition, all messages associated with this screen that
* can be displayed in the UI are defined.
*****
-->
<screen id="MyCassetteOrder" extends="PSOrder">
  <fieldGroup id="MyCassetteOrderDetails" advanced="0">
    ...
    <field id="pan" ...>
      ...
    </field>

    <field id="expirationDate" ...>
      ...
    </field>

  </fieldGroup>

  <message id="PRC=14-SRC=11010" type="error" ...>

```

```
<message id="PRC=14-SRC=11011" type="error" ...>
    ...
</screen>
```

```
</PSPL>
```

As the example shows, framework screens can be either extended or configured to add cassette-specific data to the user interface. It is important to note that not all framework screens defined in the framework's PSPL files can be extended. See "Identify and name your cassette screens" on page 128 for details.

PSPL document styles

When writing your cassette's PSPL, there are two ways to identify the text that is displayed to the end user:

- In-line in the PSPL file itself. In this situation, you must have a separate PSPL file for each language your cassette supports.
- In an associated properties file. In this situation, you have only one PSPL file, but multiple Java properties files for each language your cassette supports.

The in-line design was used for WebSphere Payment Manager Version 2.1 and is still supported for WebSphere Commerce Payments. If the in-line design is used, the cassette-specific information for a given cassette is specified in the file *cassetteName.language.PSPL*, where *cassetteName* is the name of the cassette and *language* identifies the language used for translatable text in the file.

Payment Manager Version 2.2 introduced the properties file approach for all of the framework screens. The properties file approach is available to cassette writers using Payment Manager Version 2.2 or later. In the properties file approach, screen definitions and layouts for a given cassette are contained in the cassette's PSPL file, while Properties files (one for each language) are defined to hold the actual text that is displayed on the user interface. See "Create your cassette's UI properties file" on page 131 for a list of valid language identifiers.

A general tag in the in-line designed PSPL may look like the following:

```
<tag1>Displayed Text Goes Here</tag1>
```

For the properties file based PSPL, a `textid` attribute is used as a key to look up a value in the cassette's UI properties file. The following is an example of how a general tag inside the PSPL will look.

```
<tag1 textid= "cassetteName.id.TAG1/">
```

And the corresponding properties file entry would look like this:

```
cassetteName.id.TAG1=Displayed Text Goes Here
```

The properties file approach to designing PSPL is suggested for all cassette PSPL documents and, therefore, will be discussed in greater detail than the in-line approach. The reason this is the suggested approach is that it provides for the separation of the screen definitions from the actual text that is displayed in the UI. The advantages of this are:

- If your cassette supports multiple languages, then the translation of the UI will be accomplished by translating only the properties file. In the in-line approach, the PSPL file itself must be translated. This is due to the fact that in the in-line approach, the PSPL elements and attributes are mixed in with translatable text. This makes the job of translation both difficult and error prone. If, for example,

during the course of translation any of the elements or attributes are changed by mistake, the cassette extension to the user interface will no longer work.

- If your cassette supports multiple languages, then in order to make a change to the look and feel of the cassette's UI, only one PSPL document has to change (and, optionally, the properties file(s) if a textual change is made). In the in-line approach, all look and feel changes have to be made to each language's PSPL document.

User's guide to PSPL

The following steps are provided as a guideline to be used to help write your cassette's PSPL.

1. Identify your cassette's extensions.
2. Determine which of your cassette extensions should be displayed.
3. Identify and name your cassette screens.
4. Identify and name the fields.
5. Group the fields.
6. Define the field attributes.
7. Write messages for your cassette's specific return codes.
8. Create your cassette's UI properties file(s).

Identify your cassette's extensions

Before writing the PSPL for your cassette, you should design your cassette's extensions to the framework object model. This consists of identifying those elements that extend a framework object and those that create a new object. Refer to "Identify all of your cassette's extensions" on page 160 for additional information.

Determine which of your cassette extensions should be displayed

Once you have identified your cassette's extensions, you need to determine which extensions you wish to be exposed to the end user. Refer to "Design the external view of your extensions" "Design your extensions to the framework object model" on page 160 for additional information.

Identify and name your cassette screens

Your PSPL will correlate very closely to the external view of your cassette's extension objects. In general, your PSPL file will contain the following:

- One screen definition for each of the framework objects that your cassette extends.

The framework object model has Administration objects and Payment objects. Each framework object is represented in the user interface by a screen. Your cassette may have additional data associated with these framework objects that you wish to display in the user interface. This is accomplished by the definition of a separate PSPL screen that "extends" an existing framework screen. For example, if you have a cassette-specific extension to the AccountAdmin object, you need to create a screen in your cassette's PSPL which "extends" the PSMerchantAccount screen. By extending framework screens, the cassette-specific parameters will appear seamless in the user interface. The following table shows a list of all framework screens that can be extended, their associated framework object, and a description of the data each one of these screens represent:

Framework screen name	Framework object	Represents
PSCassette	CassetteAdmin	Configuration information for a cassette
PSMerchantCassetteSettings	PaySystem	Configuration information for the association between a Merchant and a Cassette
PSMerchantAccount	AccountAdmin	Configuration information for an Account
PSOrder	Order	Order information
PSBatch	Batch	Batch information
PSPayment	Payment	Payment information
PSCredit	Credit	Credit information
PSBatchSettle	N/A	Information needed to settle a Batch
PSOrderApprove	N/A	Information needed to Approve an Order
PSOrderSale	N/A	Information needed to do a "Sale" on an Order
PSOrderRefund	N/A	Information needed to do a Refund on an Order
PSPaymentDeposit	N/A	Information needed to Deposit a Payment

- One screen definition for each new object your cassette defines in the form of a MerchantCassetteObject or SystemCassetteObject.

Your cassette may also have implemented other administrative objects that are not part of the framework object model. These objects are known as MerchantCassetteObjects or SystemCassetteObjects. In this case, it is not appropriate to extend a framework screen. Instead you will create a new screen by "configuring" a framework screen (usually the screen that is associated with the object). It is possible for a cassette to "extend" and "configure" the same screen. For example, if your cassette is credit card based, you may want to define a Brand object. Brand objects are not in the framework object model; however, they could be implemented by the cassette as a MerchantCassetteObject. Let's say your cassette wants to associate each of these cassette-defined Brand objects with an Account. In this case, you would have a screen definition that "configures" the PSMerchantAccount screen, and potentially a separate screen definition that "extends" the PSMerchantAccount screen if you have AccountAdmin extensions.

The name of each screen is specified by the id attribute of the screen tag. For screens that "extend" a framework screen, this can be any unique name but it should be meaningful. A recommended approach is to use the cassette name as the prefix and the last part of the framework screen as the suffix. For example, if you are extending the "PSOrder" screen for the "MyCassette" cassette, name the screen "MyCassetteOrder". For screens that "configure" a framework screen use the lower case name of the object for the id, for example "brand". These screens should also have an updateID with an uppercase name of the object, for example, "BRAND".

Identify and name the fields

When designing your cassette extensions to the framework object model, you decided which cassette-specific data you wanted to externalize. All cassette data that is to be externalized on the user interface must be returned by the cassette as

Query command output. There are two types of objects that a cassette can return on a Query command: a `CassetteExtensionObject` that represents cassette-specific extensions to the framework's primary object model, or a `CassetteConfigObject` that represents a cassette's own administrative object as implemented by a `MerchantCassetteObject` or a `SystemCassetteObject` (such as the `Brand` object mentioned above).

For example, say your cassette extends the `PSMerchantAccount` screen and externalizes two cassette-specific pieces of data: "accountFiled1" and "accountField2". In addition, say your cassette "configures" the `PSMerchantAccount` screen with its own "Brand" screen which contains a "brandID" attribute. When your cassette processes the `QueryAccounts` API command, it must:

- Create and populate a `com.ibm.etill.framework.cassette.query.CassetteExtensionObject` with a property id and its associated value for each cassette extension. Specifically, you would add a property for both `accountField1` and `accountField2` into the `CassetteExtensionObject`. The property id **must** match the field id in the PSPL.
- Create and populate a `com.ibm.etill.framework.cassette.query.CassetteConfigObject` with a property id and its associated value for each cassette attribute. Specifically, you would add a property for "brandID" into the `CassetteExtensionObject`. The property id **must** match the field id in the PSPL.

To summarize, when a screen is displayed, the data for the screen is provided by the XML returned by the Payment Servlet (using the `CassetteExtensionObject` and/or `CassetteConfigObject` created by your cassette's query code). In the UI, a single unit of data is represented by a `Field` element in the PSPL. In the XML a single unit of data is represented by a `CassetteProperty` element. The `propertyId` attribute of the `CassetteProperty` element must match the `id` attribute of the PSPL `Field` element.

The user interface provides support for the creation, deletion, and modification of both framework and cassette objects. For example, when you are viewing a single account in the user interface, you will see a button labeled "Update". When this button is pressed, a `ModifyAccount` API command is sent to WebSphere Commerce Payments. The parameters for this command consist of the fields of the `PSMerchantAccount` screen and any fields which you have included in your cassette's account extension. The name of these parameters come from either the field's `id` attributes in the PSPL screens, or the optionally provided `updateID` attribute. If the data on the screen is used to change values in the cassette, you will need to provide the `updateID` attribute on your cassette's fields. This is the name your cassette is expecting in the form of a "\$"-prefixed protocol data parameter on the API calls. This may be different than the field `id` parameters returned by your XML query code for output. The associated values for the parameters come from the values of the fields (which may be text boxes, drop-down lists, etc.) as entered by the user.

Group the fields

Fields can be logically grouped using the `fieldGroup` tag. You should identify related fields and place them in a field group. This provides a logical separation of information on a given screen, and provides the ability to optionally display a name, header information and trailer information for each field group on the screen. If you have a large number of fields, or some fields that are rarely configured, consider splitting the fields into multiple screens. This can be done by using the "Advanced" tag as described in the PSPL Reference section. Advanced field groups are displayed on a separate screen. Each field group requires a unique

id. Each group should have a name to identify the text to be displayed above the field group. This is specified by the name element. If you need any additional explanatory text for the group, you can specify it using the heading and trailer elements.

Define the field attributes

Once you have identified, grouped, and specified the id of each field, you need to specify the remaining field attributes. Some items influencing the field attributes are:

- Visibility of the data (e.g., passwords should be hidden)
- Whether or not the data is a single value or a choice
- The number of choices for choice data
- Whether or not multiple choices are allowed
- Whether or not the data can be modified by the user
- Whether or not the data is required to be entered by the user
- Whether or not data modification is allowed any time or only when the object is created
- Whether or not the field should be passed in an API command if it has no data

Refer to the "PSPL Field element type attribute" section "PSPL field element type attribute" on page 145 for a description of the various choices.

Each of the fields have attributes for name, shorthelp, default value, preferred width, preferred height, and maximum length. You only need to define those that you want to specify in the associated properties file. It is recommended that you specify all of these for each field so that the information can later be provided by simply modifying the properties file.

Write messages for your cassette's specific return codes

The next step is to include message elements in your cassette's PSPL for each cassette generated return code pair that is returned as a result of your cassette processing an API command. Message elements can be defined within the PSPL element, within Screen elements, within FieldGroup elements, or within Field elements. If a message can be associated with multiple screens, define it within the PSPL element. If a message is associated with multiple FieldGroup elements, define it within the Screen element. If a message is associated only with Fields within a FieldGroup, define it within the FieldGroup element. If a message is unique to a Field, define it within the Field element. Messages require an id tag that indicates the primary return code and the secondary return code. For example, <message id="PRC=14-SRC=22001">. Return codes identified by the framework will still work on cassette extended screens. Messages should be classified as an error, a warning, or informational.

Create your cassette's UI properties file

As mentioned earlier, the properties file approach for PSPL documents was introduced in WebSphere Payment Manager Version 2.2. If your cassette is utilizing this style of PSPL, then a cassette UI properties file must be defined for each language your cassette supports. Each properties file contains the cassette-specific text that is displayed on the UI. The naming convention for these files is `cassetteNameUI_language.properties`

where *cassetteName* is the name of the cassette and *language* identifies the language used for translatable text in the file. The file `cassetteNameUI.properties` (without a language identifier) will also need to be created as a base reference for text if a user of an unsupported language tries to use the WebSphere Commerce Payments

user interface. Usually this base properties file is a copy of one of the other language's properties file. This provides a convenient way to show the user interface in a default language for those languages your cassette does not support. For example, in the WebSphere Commerce Payments framework it is a copy of the English properties file.

The *textid* attribute is the most important aspect of properties-based PSPL. The textid specified in the PSPL must match the key in the properties file. This textid in the PSPL file must be unique so that there are no duplicate keys in the properties file. To meet these requirements, there is a textid naming convention that will make each textid unique and easy to find in the PSPL document.

There are four levels of textids that go into the construction of a unique name. The first is the component level. This defines content that can be seen on all the cassette-specific panels in the WebSphere Commerce Payments user interface. The component is then made up of screens. The screen level defines attributes that can be seen in the user interface on one panel. Fieldgroup is the next level. This identifies a group of fields in the user interface. The lowest level is the field. This level identifies attributes per cassette field in the user interface.

Below is the naming convention for textid's in WebSphere Commerce Payments. Note that anything inside brackets needs to be substituted by the actual value of the identifiers you will create in the PSPL document. Refer back to this table when writing your PSPL.

Level	textid naming convention
Component	<cassetteName>.shortHelp <cassetteName>.NAME <cassetteName>.HEADER <cassetteName>.TRAILER <cassetteName>.HELPPFILE <cassetteName><messageID>.MESSAGE
Screen	<cassetteName>.<screenID>.NAME <cassetteName>.<screenID>.HEADER <cassetteName>.<screenID>.TRAILER <cassetteName>.<screenID>.EMPTYLIST <cassetteName>.<screenID>.<messageID>.MESSAGE* <cassetteName>.<screenID>.<actionID>.ACTION
Field Group	<cassetteName>.<screenID>.<fieldgroupID>.NAME <cassetteName>.<screenID>.<fieldgroupID>.HEADER <cassetteName>.<screenID>.<fieldgroupID>.TRAILER
Field	<cassetteName>.<screenID>.<fieldgroupID>.<fieldID>.NAME <cassetteName>.<screenID>.<fieldgroupID>.<fieldID>.SHORTHELP <cassetteName>.<screenID>.<fieldgroupID>.<fieldID>.DEFAULTVALUE <cassetteName>.<screenID>.<fieldgroupID>.<fieldID>.PREFERREDWIDTH <cassetteName>.<screenID>.<fieldgroupID>.<fieldID>.PREFERREDHEIGHT <cassetteName>.<screenID>.<fieldgroupID>.<fieldID>.MAXIMUMLLENGTH <cassetteName>.<screenID>.<fieldgroupID>.<fieldID>.<optionID>.OPTION

*messageID attributes can sometimes contains the "=" character. When this happens, the "=" should be left out of the textid because of the problems it can create in properties files. For example, a message element that has a message id of "PRC=4-SRC=617" should use "PRC4-SRC617" when constructing the textid. For example:


```
<message id=" PRC=4-SRC=617 " type="error"
textid="admin.PSMerchantAccount. PRC4-SRC617 .MESSAGE"/>.
```

There are three additional keys that must be added to every cassette user interface properties file. They are:

- The cassette's help file, as specified by the key `<cassetteName>.helpFile`. The help file, must be set to the location of the help file located under the web publish directory.
- The Encoding, as specified by the key `encoding`. The encoding must be set to the code page of the properties file language locale. Below is a table of countries/regions, locales, and corresponding encodings to use in your properties file:

Country/region	Locale	Encoding
Germany	de	UTF-8
Spain	es	UTF-8
US	en	UTF-8
France	fr	UTF-8
Italy	it	UTF-8
Japan	ja	UTF-8
Korea	ko	UTF-8
Brazil	pt	UTF-8
Chinese Simplified	zh	UTF-8
Chinese Traditional	zh_TW	UTF-8

- The list of supported languages (a mandatory value that must be written in the UI properties file). You must create a `cassetteNameUI.Lang=Language_locales_separated_by_commas` property in the base cassette UI properties file. For example, if the cassette named "MoneyCard" supported English, French, Japanese, and Korean languages, the property would look like `MoneyCardUI.Lang=en,fr,ja,ko` in the base `MoneyCardUI.properties` file.

PSPL reference

A PSPL document is made up of elements, each of which is composed of attributes and other elements. The following sections describe those elements and attributes used by cassette developers. Review of the `pspl.dtd` will show additional attributes and elements. Those elements are used primarily by the framework's PSPL files and are not described here.

Some of the attributes are only used for properties based PSPL, some are only used for in-line PSPL, and most are common to both. Attributes and elements apply to both unless otherwise indicated. All attributes and elements are optional unless otherwise indicated.

The examples are based on the OfflineCard Cassette and CustomOffline Cassette. In the examples shown, many of the tag names are split onto separate lines. This is done for readability. These tag names should be coded on a single line.

The attributes and elements are described in tabular form using the following conventions:

- Name - the name of the attribute or element

- Description - a description of the element
- R/O - an indication of whether the attribute or element is required (R) or optional (O).

PSPL element

The PSPL *element* is the top-level element in every PSPL document. Its attributes include:

Name	Description	R/O
component	Identifies the cassette to the framework. Must match the name of your cassette.	R
locale	Identifies the language for the HTML page. Only used for in-line PSPL. For properties based PSPL it should be coded in the associated properties file.	O
encoding	Identifies the character encoding of the HTML page. Only used for in-line PSPL. For properties based PSPL it should be coded in the associated properties file.	O
helpFile	Specifies the document containing context-sensitive help for the cassette. Only used for in-line PSPL. For properties based PSPL it should be coded in the associated properties file.	O

In addition to the above attributes, the PSPL element in a cassette PSPL document contains the elements which identify extensions to framework user interface screens as well as other structural information. These elements include:

Name	Description
name	Identifies the cassette name to be displayed on the cassette specific screens that have been reached via the Merchant Setting navigation menu.
vendor	Specifies the name and URLs of the cassette vendor. This information is displayed on the PMCassette screen. The vendor element attributes are described in PSPL element vendor attributes.
shortHelp	Contains a brief description of the cassette which is displayed on the PMCassette screen.
header	Contains text or other well formed HTML elements to display at the top of cassette-specific user interface screens.
trailer	Element contains text or other well formed HTML elements to display at the bottom of cassette-specific user interface screens.
screen	List of screens describing either a new object, or extensions to a related framework object. Refer to the "PSPL Screen element" section.
message	List of messages to be displayed in certain circumstances. These are messages that apply to more than one of the defined screens. If the message is unique to a screen, it should be defined with that screen.

The vendor element describes the vendor data to be displayed. Attributes are:

PSPL element vendor attributes

Name	Description
URL	A link to the vendor's page.

PSPL element vendor attributes

Name	Description
brandURL	A link to an image to be displayed as part of the header on the cassette specific screens that have been reached via the Merchant Settings navigation menu. If not specified, the framework defined image is shown.
textid	The key into the properties file. Only used for properties based PSPL.

Figure 5 shows the WebSphere Commerce Payments Cassette screen with the location of some of these elements. Note that the image displayed is the framework's image, and not the cassette specified image. Because this is the framework screen, the image is the framework's image. If this were one of the screens reached via the Merchant Settings navigation menu, the image would be the cassette's defined image.

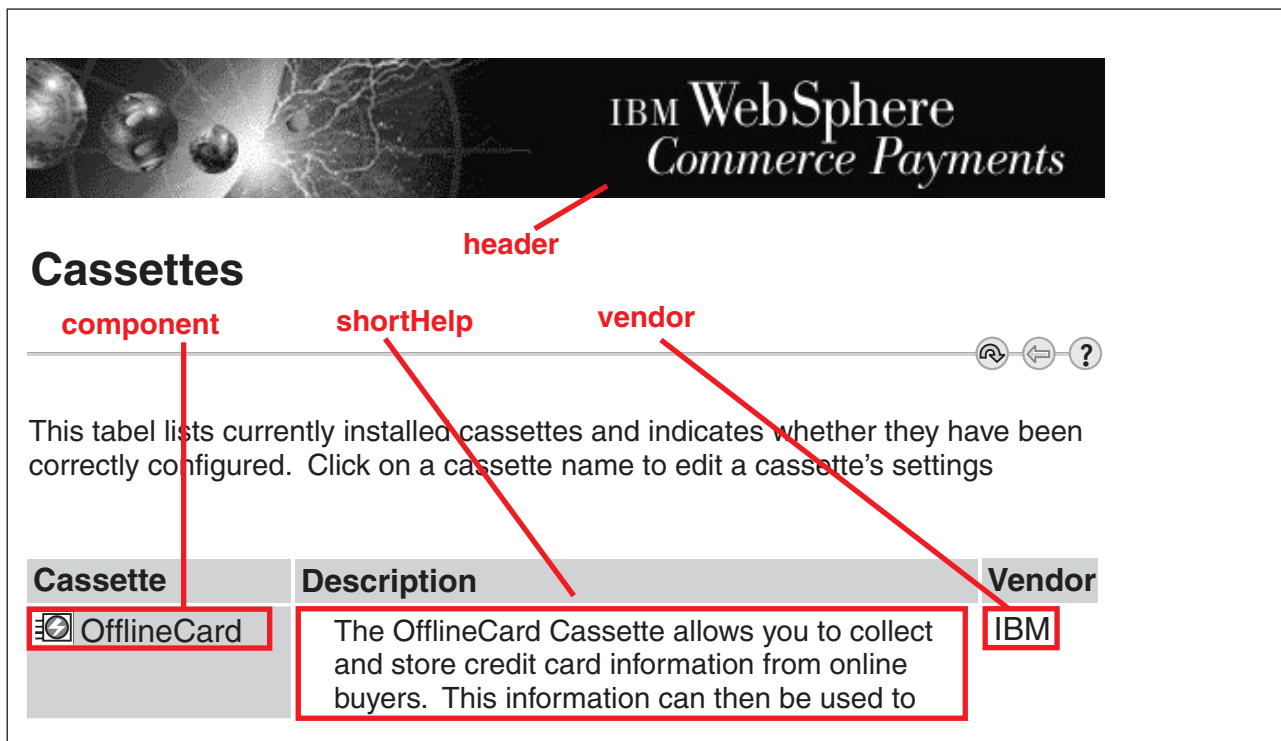


Figure 5. Element location

Properties based example:

- PSPL file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE PSPL SYSTEM "pspl.dtd">
<PSPL component="OfflineCard">
  <name textid="OfflineCard.NAME"/>
  <header textid="OfflineCard.HEADER"/>
  <shortHelp textid="OfflineCard.SHORTHELP"/>
  <trailer textid="OfflineCard.TRAILER"/>
  <vendor URL="http://www.ibm.com/payment"
    brandURL="/webapp/PaymentManager/images/ibm.gif">IBM</vendor>
  ...
</PSPL>
```

- Properties file:

```

encoding=UTF-8
OfflineCardUI.Lang=de,en,es,fr,it,ja,ko,pt,zh,zh_TW
OfflineCard.helpFile=/webapp/PaymentManager/{0}/OfflineCardframe.html
OfflineCard.NAME=OfflineCard Cassette
OfflineCard.HEADER=<IMG SRC="/webapp/PaymentManager/images/paymgr.gif"
BORDER="0" ALT="IBM WebSphere Commerce Payments">
OfflineCard.SHORTHELP=The OfflineCard Cassette allows you to collect \
and store credit card information from online buyers. \
This information can then be used to ..... \
... \
Cassette documentation is provided in \
<a target="helpFrame"
href="/webapp/PaymentManager/{0}/paymgradmin.html#HDROFFCARDSUP">HTML</a>.

```

In-line example

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE PSPL SYSTEM "pspl.dtd">
  <pspl component="OfflineCard">
    <locale="en">
      <encoding="ISO-8859-1">
        <helpFile="/webapp/PaymentManager/en/OfflineCard.html">
          <styleSheet="/webapp/PaymentManager/style.css">
            <name>OfflineCard Cassette</name>
            <vendor URL="http://www.ibm.com/payment">
              <brandURL="/webapp/PaymentManager/images/ibm.gif">IBM</vendor>
            <shortHelp>The OfflineCard Cassette defines a credit card payment
              cassette for testing and example purposes. </shortHelp>
            <header><![CDATA[<IMG SRC="/webapp/PaymentManager/images/paymgr.gif"
              BORDER="0" ALT="IBM WebSphere Commerce Payments"></header>
            ...
          </pspl>

```

PSPL — screen elements

Screen elements allow cassettes to specify protocol-specific content and layout. A PSPL document may contain multiple screen elements which control how objects are displayed in the user interface. A given screen may either "extend" a framework screen or "configure" a framework screen.

- "Extends" is used when there is a one to one relationship between the cassette screen and the framework screen. The underlying cassette object provides additional attributes for the framework object (e.g., additional field for an account). The information from the framework screen is shown, followed by the information from the cassette screen. Figure 6 on page 137 shows an example of an "extended" screen. The data above the "Account Details" group heading, and the buttons, is framework data. The "Account Details" group heading is cassette data.

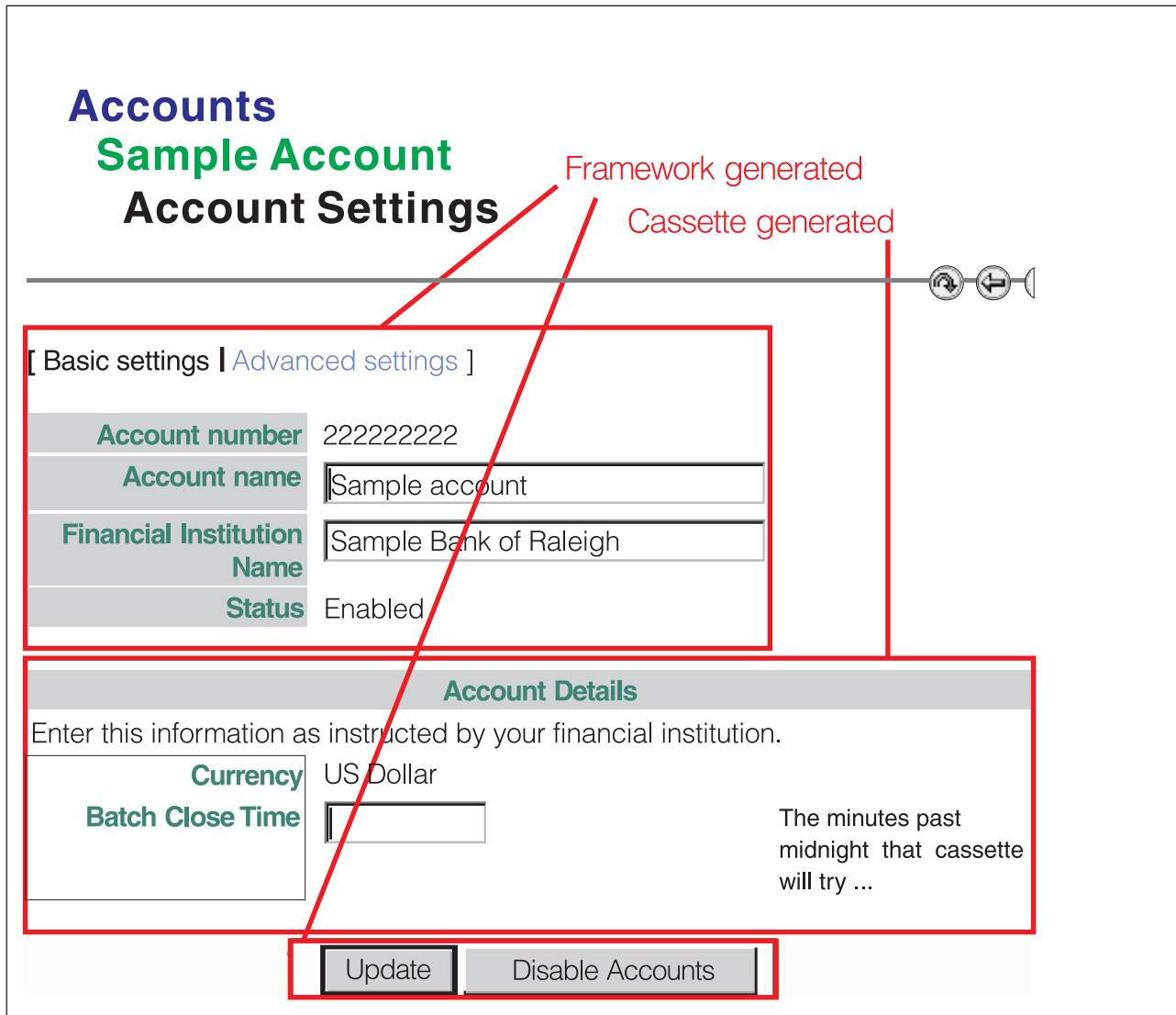


Figure 6. Extended screen

- "Configures" is used when there is a many to one relationship between the cassette screen and the framework screen. This occurs when the cassette has defined additional objects which are associated with the framework object (eg brands for an account). The framework automatically modifies the associated framework screen with a reference to the screen being configured and displays an intermediate screen to list the defined objects as shown in Figure 7 on page 138 and Figure 8 on page 138. The actual content is displayed on a separate screen as shown in Figure 9 on page 139.

Accounts

Sample account

Settings	Description
Account Settings	Create, update or delete accounts
Brands	Add or delete brands for this account.

Added by framework

Figure 7. Framework-generated screen #1

Accounts

Sample Account

Brands

Click on a brand to view its current status.

Brand Name	
<input type="checkbox"/>	Silver Card
<input type="checkbox"/>	Gold Card
<input type="button" value="Add a Brand..."/>	<input type="button" value="Delete Selected Brands..."/>

Framework generated

Figure 8. Framework-generated screen #2

Accounts
Sample account
Brands
Silver Card

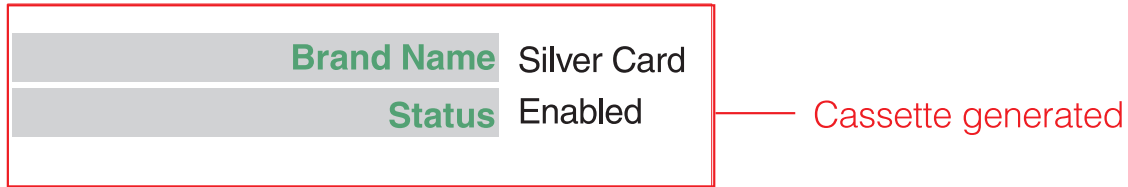


Figure 9. Content screen

A screen element has the following attributes:

Name	Description	R/O
id	Acts as a identifier for the screen. The value must be unique and is used as a reference to the properties file (properties based PSPL) containing screen text.	R
updateID	Identifies the Merchant Cassette object associated with this screen. Must match the name in you cassette code. This attribute is used when screens are configured.	O
extends (see note)	Indicates that a screen in the cassette PSPL has a one-to-one relationship with the framework screen and describes the additional data to be shown. The value of this attribute is the id of the framework screen being extended.	O
configures (see note)	Indicates that a screen in the cassette PSPL has a many-to-one relationship with a framework screen such that an intermediate screen is displayed to list the defined objects. The value of this attribute is the id of the framework screen being configured.	O

Note: Either extends or configures is used but not both. One of them is required. A given screen may be both extended and configured. To do this, two screens are defined in the cassette's PSPL.

In addition to the above attributes, screens can contain zero or more of the following elements:

Name	Description
name	Title to be displayed at the top of the page.
header	Information to be displayed after the name but before the data. This can be any well formed HTML tags such as a paragraph, table, or list.
trailer	Information to be displayed after the name but before the data. This can be any well formed HTML tags such as a paragraph, table, or list.

Name	Description
emptyList	Text to be displayed when no data is returned.
shortHelp	Text to be displayed to the right of the screen when it is shown in a list.
fieldGroup	See "PSPL - fieldGroup element" on page 141 for information on logical groupings of individual fields.
action	See "PSPL - action element" on page 150 for information on buttons that appear in the user interface.
message	See "PSPL - message element" on page 152 for information on messages specific to this screen.

Examples showing the 'OfflineCardAccount' describing a cassette extension to the AccountAdmin object which extends the 'PSMerchantAccount' screen.

Properties Based example:

- PSPL file:

```
<screen id="OfflineCardAccount" extends="PSMerchantAccount" >
<name textid="OfflineCard.OfflineCardAccount.NAME"/>
<header textid="OfflineCard.OfflineCardAccount.HEADER"/>
<shortHelp textid="OfflineCard.OfflineCardAccount.SHORTHELP"/>
<trailer textid="OfflineCard.OfflineCardAccount.TRAILER"/>
<emptyList textid="OfflineCard.OfflineCardAccount.EMPTYLIST"/>
<fieldGroup id="OfflineCardAccountDetails" ...>
...
</fieldGroup>
```

Associated properties file:

```
OfflineCard.OfflineCardAccount.NAME=Account Settings
OfflineCard.OfflineCardAccount.SHORTHELP=Create, update or delete accounts.
```

Note: In the above example, not all fields in the PSPL file have an entry in the properties file. Only those that have an entry in the file will be displayed. The additional fields in the PSPL file are there in case the merchant wants to add customization text via the properties file.

In-line format example:

```
<screen id="OfflineCardAccount" extends="PSMerchantAccount">
  <name>Account Settings</name>
  <shortHelp>Create, update or delete accounts.</shorthelp>
  <fieldGroup id="OfflineCardAccountDetails" ... >
    ...
  </fieldGroup>
  ...
  <message id="PRC=3-SRC=1060" ...</message>
  ...
</screen>
```

Examples showing the 'brand' screen describing a cassette-defined MerchantCassetteObject which configures the 'PSMerchantAccount' screen:**Properties Based example:**

- PSPL File

```
<screen id="brand" updateID="BRAND" configures="PSMerchantAccount">
  <name textid="OfflineCard.brand.NAME"/>
  <header textid="OfflineCard.brand.HEADER"/>
  <shortHelp textid="OfflineCard.brand.SHORTHELP"/>
  <trailer textid="OfflineCard.brand.TRAILER"/>
  <emptyList textid="OfflineCard.brand.EMPTYLIST"/>
  <fieldGroup id="brand">
```



```

    ...
  </fieldGroup>
  ...
</screen>

```

- Associated properties file:

```

# Screen: OfflineCard.brand
OfflineCard.brand.NAME=Brands
OfflineCard.brand.HEADER=Click on a brand to view its current status.
OfflineCard.brand.SHORTHELP=Add or delete brands for this account.
OfflineCard.brand.EMPTYLIST= To create a brand, click on Add a Brand.

```

In-line format example:

```

<screen id="brand" updateID="BRAND" configures="PSMerchantAccount">
  <name>Brands</name>
  <shortHelp>Add, edit, or delete brands for this account.</shorthelp>
  <header><![CDATA[Click on a brand to edit or delete the brand.</header>
  <emptyList>No brands exist for this account. Create a brand by clicking
    Add a Brand.</emptyList>
  <fieldGroup id="brand">
    ...
  </fieldGroup>
</screen>

```

PSPL - fieldGroup element

A *fieldGroup* is a logically-related collection of fields within a screen. Field groups can be configured to display on a single page or on multiple pages which are linked together. There is one page for "basic settings" and one for "advanced settings" When multiple pages are linked, the links are placed at the top of the page as shown in Figure 10 below:



Figure 10. fieldGroup element, advanced settings

The fieldGroup element has the following attributes:

Name	Description	R/O
id	A unique identifier.	R
advanced	Indicates how groups are to be displayed. If not provided, all groups are displayed on a single page. If provided, the groups are split between two pages. Valid values are 0 or 1 where 0 identifies the "basic settings" while "1" identifies the "advanced settings".	O

In addition to the above attributes, a fieldGroup may contain zero or more of the following elements:

Name	Description
name	Text that will be displayed in the user interface centered in a shaded area above the group's fields.
header	Text, or other well formed HTML elements, that is displayed left aligned above the fields in the group.
trailer	Text, or other well formed HTML elements, that is displayed left aligned below the fields in the group
message	One or more messages associated with this group. Refer to the "PSPL field element type attribute" on page 145 for additional information.
field	A list of 'field' elements to be displayed as a group. Refer to the "PSPL field element type attribute" on page 145 for additional information.

Properties based example

- PSPL file:

```
<screen id="OfflineCardAccount" extends="PSMerchantAccount" >
  ...
  <fieldGroup id="OfflineCardAccountDetails" advanced="0">
    <name textid="OfflineCard.OfflineCardAccount.
      OfflineCardAccountDetails.NAME"/>
    <header textid="OfflineCard.OfflineCardAccount.
      OfflineCardAccountDetails.HEADER"/>
    <trailer textid="OfflineCard.OfflineCardAccount.
      OfflineCardAccountDetails.TRAILER"/>
    <field id="Currency" ...>
      ...
    </field>
    <field id="BatchCloseTime" ... >
      ...
    </field>
  </fieldGroup>
  ...
</screen>
```

- Associated properties file:

```
# Screen: OfflineCard.OfflineCardAccount
...
OfflineCard.OfflineCardAccount.OfflineCardAccountDetails.HEADER=
Enter this information as instructed by your financial institution.
OfflineCard.OfflineCardAccount.OfflineCardAccountDetails.NAME=
Account Details
...
```

In-line example

- PSPL file:

```
<screen id="OfflineCardAccount" extends="PSMerchantAccount">
  ...
  <fieldGroup id="OfflineCardAccountDetails" advanced="0">
    <name>OfflineCard Account Details.</NAME>
    <header ![CDATA[Enter this information as instructed
      by your financial institution.</i>]].</HEADER>
    <field id="Currency" ...>
      ...
    </field>
  </fieldGroup>
  ...
</screen>
```

PSPL - field element

A *field element* partially defines how a payment server datum can be presented. Each field element contains the following attributes:

Name	Description	R/O
id	The unique identifier for this field which must match the name specified in your cassette code.	R
updateID	Identifies the protocol specific keyword for the data being updated. Must match the name in your cassette code.	O
type	The type of information described by the field (e.g., text, date, or amount). These types are described in "PSPL field element type attribute" on page 145.	O
columnPosition	Defines the relative position of the element on the panel when displayed in tabular format. The fields are laid out in ascending order based on the number.	O
sortOrder	Defines the relative position of the element for sorting purposes when displayed in tabular format. The numbers represent the sorting significance of the field with the lowest value representing the most significant sort field.	O
sortDirection	Indicates whether the fields are sorted in ascending or descending sequence. 0 indicates ascending, 1 indicates descending.	O
linkField	Indicate that this file is to be displayed as a link. Clicking on the link causes the details screen for the object to be displayed.	O
keyField	Used on screens that "configure" framework screens to identify the field that represents a key into the database. Valid values are "full" and "partial". Use "full" if the field, in combination with the keys of the corresponding framework object, allows for selection of a unique object (e.g. Brand). When Brand is combined with Merchant and Account from the framework, a unique object is identified. Use "partial" if the field does not allow for selection of a unique object, For example, if there was a many to one relationship of some object to a brand (e.g. issuing bank), the brand would have a "partial" key. When Brand is combined with Merchant and Account from the framework, a unique object is not "issuing bank" field to identify the object.	O
sendEmptyValue	Indicates whether or not a field that has not been filled in is returned. 0 indicates no, 1 indicates yes.	O
required	Indicates whether or not the field is required. Valid values are 0 and 1. If set to 1, a red asterisk is displayed to the left of the field.	O

Name	Description	R/O
multipleValues	Identifies whether or not multiple selections are allowed. A "0" indicates a single selection while a "1" indicates multiple selections. This, in conjunction with the displayList attribute, determines how the field is shown. A "0" indicates the use of either a radio button or a drop down list. A "1" indicates the use of either a check box or a list box.	O
displayList	Indicates whether or not this field represents a list. A "0" indicates no and a "1" indicates yes. This, in conjunction with the multipleValues attribute, identifies how the associated options are displayed. A "0" indicates the use of either a check box or a radio button, based on the value of the multipleValues tag. A "1" indicates the use of a list box or drop down list, based on the value of the multipleValues tag. See Table 9 on page 147 for more information on displayList values.	O
columnJustification	Indicates how text is displayed in the field. Valid values are "left", "right", "centered".	O
displayType	Indicates whether or not data can be entered in the field.	O

Whether the fields are laid out vertically (details view) or horizontally (tabular view) is controlled by the code. In general queries result in a tabular view while access to a particular object (e.g., account 1234) result in a details view.

In addition to the above attributes, a Field can also contain zero or more of the following elements:

Name	Description
name	Text that will be displayed in a shaded area to the left of the fields data in detail form, or above the column in tabular form.
shortHelp	The text to be displayed to the right of the field.
columnWidth	The width of the data column when data is presented in tabular form.
preferredWidth	The width of the field in pixels or percent.
preferredHeight	The height of the field in pixels or percent.
maxLength	The maximum number of characters allowed in the field.
defaultValue	The data used to initialize the field. This is either a text value or an index into an options list.
option	List of options associated with the field as described in "PSPL - option element" on page 149.
message	List of messages associated with this field as described in "PSPL - message element" on page 152.

Properties based example

- PSPL File:

```

<screen id="OfflineCardAccount" extends="PSMerchantAccount">
  ...
  <fieldGroup id="OfflineCardAccountDetails" advanced="0">
    ...
    <field id="BatchCloseTime" updateID="$BATHCLOSETIME"
      sendEmptyValue="1" type="text" displayType="readWrite">
      <name textid="OfflineCard.OfflineCardAccount.
        OfflineCardAccountDetails.BatchCloseTime.NAME"/>
      <shortHelp textid="OfflineCard.OfflineCardAccount.
        OfflineCardAccountDetails.BatchCloseTime.SHORTHELP"/>
      <defaultValue textid="OfflineCard.OfflineCardAccount.
        OfflineCardAccountDetails.BatchCloseTime.DEFAULTVALUE"/>
      <columnWidth textid="OfflineCard.OfflineCardAccount.
        OfflineCardAccountDetails.BatchCloseTime.COLUMNWIDTH"/>
      <preferredWidth textid="OfflineCard.OfflineCardAccount.
        OfflineCardAccountDetails.BatchCloseTime.PREFERREDWIDTH"/>
      <preferredHeight textid="OfflineCard.OfflineCardAccount.
        OfflineCardAccountDetails.BatchCloseTime.PREFERREDHEIGHT"/>
      <maxLength textid="OfflineCard.OfflineCardAccount.
        OfflineCardAccountDetails.BatchCloseTime.MAXIMUMLength"/>
    </field>
  </fieldGroup>
</screen>

```

- Associated properties file:

```

OfflineCard.OfflineCardAccount.OfflineCardAccountDetails.
BatchCloseTime.NAME=
Batch Close Time
OfflineCard.OfflineCardAccount.OfflineCardAccountDetails.
BatchCloseTime.SHORTHELP=
The minutes past midnight that cassette will try \
to automatically close open batches. A value of \
0 (zero) represents midnight and 1439 is maximum value \
allowed. A null value disables automatic batch closing.
OfflineCard.OfflineCardAccount.OfflineCardAccountDetails.
BatchCloseTime.PREFERREDWIDTH=10
OfflineCard.OfflineCardAccount.OfflineCardAccountDetails.
BatchCloseTime.MAXIMUMLength=10

```

PSPL field element type attribute

This attribute defines how the field is displayed. The valid types are:

Name	Description
text	A standard text input field.
untrimmedText	An input field used if your cassette needs data that has leading or trailing spaces. The "text" field type truncates any white spaces before the start of the text and any white spaces after the text, whereas the "untrimmedText" accepts exactly what is entered by the user.
integer	A text field limited to numeric values.
date	A field that accepts a date as input in a format specific to the operating system locale. For English, the correct format would be: mm/dd/yy. A field that has this type and does not have shortHelp will be given an example date in the shortHelp section by default.
dateTime	Text field that displays receives a timestamp and displays it as a date, based on the locale.
password	A standard text input field that shows asterisks (*) instead of text.

Name	Description
messageID	<p>A text field that displays a message based on the field content. The message associated with a given content is defined in the properties file. For example, the order state is defined in the PSPL file as:</p> <pre data-bbox="665 315 1356 399"><field id="state" type="messageID" displayType="readOnly"> ... </field></pre> <p>The associated properties file contains:</p> <pre data-bbox="665 462 1234 640">payment.stateorder_requested.MESSAGE=Requested payment.stateorder_ordered.MESSAGE=Ordered payment.stateorder_refundable.MESSAGE=Refundable payment.stateorder_rejected.MESSAGE=Rejected payment.stateorder_pending.MESSAGE=Pending payment.stateorder_cancelled.MESSAGE=Cancelled payment.stateorder_closed.MESSAGE=Closed</pre> <p>If the cassette supports independent credit, the file does not contain <code>payment.stateorder_ordered.MESSAGE=Ordered</code>.</p>
boolean	A text input field in the form of a check box.
currency	A field that shows a drop down selection list of currencies defined by the framework.
merchant	A field that appears as either a standard input field or a list field, depending on the number of merchants.
cassette	A list of cassette names.
amount	Text input field limited to numeric data and which automatically displays default short help if none is provided.
account	A field that appears as either a standard input field or a list field, depending on the number of merchants.
status	<p>A text field that displays the status of an administration object based on the combination of the objects pending, enabled, active, and valid flags as shown in the following table. The result string determines the text of the message to be displayed. The returned value consists of the result string prefixed by "xxx=" where xxx is the field ID. See Table 10 on page 147 for the order in which tests are made. See Figure 11 on page 147 for an example of the property based PSPL to show the status.</p>
sortedSelection	A selection list containing the values defined by the options element which is sorted into ascending order.
customSelection	A field that acts as a text field by default but can be made to act as a selection list by defining values in the associated properties file.
fieldOrder	Defines the order in which the fields are displayed on the screen. This is only valid for properties based PSPL in version 2.2.0.1 or later.
selection	An unsorted selection list containing the values defined by the options element.
URL	<p>Displays a link to the value contained by this field. If your cassette returns a value that is always a URL, this type of field displays a clickable link instead of displaying the URL in the user interface. The link that is created is shown by a small arrow in this field. This image can be changed by setting <code>payment.IURLICON.MESSAGE</code> to point to a new image in the <code>PMCustomUI.properties</code> file.</p>

Table 9. Display List Values

displayList	multipleValues	how shown
0	0	Radio button
0	1	Check Box
1	0	Drop down list
1	1	List Box

Table 10. Status Processing Values

pending	enabled	active	valid	result string
—	—	—	No	invalid
—	—	No	Yes	stopped
—	No	Yes	Yes	stopping
Yes	Yes	Yes	Yes	pending
No	Yes	Yes	Yes	started

```

<field id="brandStatus" type="status" >
    <name textid="OfflineCard.brand.brand.brandStatus.NAME"/>
    ...
</field>
<message id="brandStatus=started" type="info"
    textid="OfflineCard.brand.brand.brandStatusstarted.MESSAGE"/>
<message id="brandStatus=pending" type="info"
    textid="OfflineCard.brand.brand.brandStatuspending.MESSAGE"/>
<message id="brandStatus=stopped"
    type="info" textid="OfflineCard.brand.brand.brandStatusstopped.MESSAGE"/>
<message id="brandStatus=invalid"
    type="info" textid="OfflineCard.brand.brand.brandStatusinvalid.MESSAGE"/>

```

Figure 11. Status Example

PSPL - displayType attribute

The *displayType* attribute defines whether or not data may be entered in the field. The following display types are used:

Name	Description
createOnly	Data for this field is editable at object creation, but will be displayed as readOnly after that time.
readOnly	Data for this field will be displayable only (not editable).
readWrite	Data for this field will always be editable.
hidden	Data for this field will not be displayed through the UI.

This example shows a definition of a property based read-write field:

```

<field id="BatchCloseTime" updateID="$BATCHCLOSETIME"
    type="text" displayType="readWrite">
    <name textid="OfflineCard.OfflineCardAccount.OfflineCardAccountDetails.
        BatchCloseTime.NAME"/>
    ...
</field>

```

PSPL - Name element

The *name* element identifies text to be displayed. Its attributes are:

Name	Description	R/O
textid	Identifies the key used to access the text in the PSPL file. Only used for properties based PSPL.	O

PSPL - Header element

The *header* element identifies text to be displayed before the data. Its attributes are:

Name	Description	R/O
textid	Identifies the key used to access the text in the PSPL file. Only used for properties based PSPL.	O

PSPL - Trailer element

The *trailer* element identifies text to be displayed after the data. Its attributes are:

Name	Description	R/O
textid	Identifies the key used to access the text in the PSPL file. Only used for properties based PSPL.	O

PSPL - shortHelp element

The *shortHelp* element identifies text to be displayed with a field. Its attributes are:

Name	Description	R/O
textid	Identifies the key used to access the text in the PSPL file. Only used for properties based PSPL.	O

PSPL - emptyList element

The *emptyList* element identifies text to be displayed when no data is returned. Its attributes are:

Name	Description	R/O
textid	Identifies the key used to access the text in the PSPL file. Only used for properties based PSPL.	O

PSPL - fieldPosition element

The *fieldPosition* element defines the position of a field in relation to other fields in the same field group in which the field is displayed on the screen. A field can contain zero or one fieldPosition element, with a value of 0 to 39. A fieldPosition value greater or equal to 40 does not display the field on the screen. This attribute may be useful when your cassette needs to display a different order of fields depending on a language locale. For example, some regions display addresses in a different order than others. In this case, the locale's property file indicates the order in which the fields should appear.

Its attributes are:

Name	Description	R/O
textid	Identifies the key used to access the text in the PSPL file. Only used for properties based PSPL.	O

PSPL - preferredWidth element

The *preferredWidth* element identifies the width of the data to be displayed. Its attributes are:

Name	Description	R/O
textid	Identifies the key used to access the text in the PSPL file. Only used for properties based PSPL.	O

PSPL - preferredHeight element

The *preferredHeight* element identifies the height of data to be displayed. Its attributes are:

Name	Description	R/O
textid	Identifies the key used to access the text in the PSPL file. Only used for properties based PSPL.	O

PSPL - maximumLength element

The *maximumLength* element identifies the length of data to be entered. Its attributes are:

Name	Description	R/O
textid	Identifies the key used to access the text in the PSPL file. Only used for properties based PSPL.	O

PSPL - defaultValue element

The *defaultValue* element identifies text to be displayed before the data. Its attributes are:

Name	Description	R/O
textid	Identifies the key used to access the text in the PSPL file. Only used for properties based PSPL.	O

PSPL - columnWidth element

The *columnWidth* element identifies the width of column used to display text. Its attributes are:

Name	Description	R/O
textid	Identifies the key used to access the text in the PSPL file. Only used for properties based PSPL.	O

PSPL - option element

An *option* element defines the choices for a selection field. These values are shown in a list, as the label of a radio button, or as the label of a check box based on the *type* attribute of the field. Option elements have the following attributes:

Name	Description	R/O
id	Unique identifier for the option which is passed to WebSphere Commerce Payments on the request.	R
default	A flag indicating that this option is to be chosen when the page is loaded. Valid values are "0" and "1" with "1" indicating that this is the default.	O

Name	Description	R/O
textid	Properties file key of the text to be displayed. This attribute is only used for properties based PSPL.	O

Properties based example

- PSPL File:

```
<screen id="currency">
  ...
  <fieldGroup id="currency" type="currency">
    <option id="004" textid="payment.currency.currency.currency.004.OPTION">
      ...
      <option id="840" textid="payment.currency.currency.currency.840.OPTION">
        </field>
      ...
    </fieldGroup>
  </screen>
```

- Associated properties file:

```
...
payment.currency.currency.currency.004.OPTION=Afghanistan Afghani
...
payment.currency.currency.currency.840.OPTION=US Dollar
```

PSPL - action element

An *action* element defines a button that submits a form. It is only used on those screens that "configure" a framework screen. For those screens that "extend" a framework screen, the actions are defined on the framework screen. There are two types of actions to consider:

- Standard actions provided on the framework screen being "configured". The framework has certain standard actions which must be defined by the cassette. These are:
 - add
 - create
 - delete
 - update
- Cassette unique actions. These are any additional actions needed by your cassette.

Action elements have the following attributes:

Name	Description	R/O
id	Unique identifier for the action. For standard framework actions, the valid values are "add", "create", "delete", and "update." For cassette unique actions, the value is whatever is recognized by your cassette.	R
updateID	Identifier to be passed in on the request. When this value is omitted, it defaults to the id value. For framework actions this should be omitted. For cassette unique actions, this should be the protocol data keyword recognized by your cassette(eg \$CONTINUE).	O

Name	Description	R/O
default	A flag indicating that this button is the default button which will get the input focus when the page is loaded. Valid values are "0" and "1", with "1" indicating the button should get the focus.	O
safe	Indicates that clicking this button does not update the database and therefore the screen will be placed in the navigation history. Clicking back on the subsequent screen will return you to this window. If a button is not safe, the this screen is not placed in the history. Clicking back on the subsequent screen will return you the window that invoked this window. This is to prevent double updates of the database. Buttons that only navigate to another screen (eg add) should be designated as safe. Buttons that initiate a query may also be designated as safe. Valid values are "0" and "1".	O
sendNoData	Indicates that data from those fields containing a updateID will not be sent with this button. Valid values are "0" and "1". This could be used in the case of an abort or cancel request.	O
textid	The text to be displayed on the button. Only used on properties based PSPL.	O

Properties based example:

- PSPL file:

```
<screen id="brand" updateID="BRAND" configures="PSMerchantAccount">
  ...
  ...
  <action id="add" safe="1" textid="admin.PSMerchantAccount.add.ACTION"/>
  <action id="create" textid="admin.PSMerchantAccount.create.ACTION"/>
  <action id="delete" textid="admin.PSMerchantAccount.delete.ACTION"/>
  <action id="update" textid="admin.PSMerchantAccount.update.ACTION"/>
  <action id="cancel" updateID="$CANCEL" sendNoData="1"
    textid="admin.PSMerchantAccount.cancel.ACTION"/>
</screen>
```

- Associated properties file:

```
admin.PSMerchantAccount.add.ACTION=Add an Account...
admin.PSMerchantAccount.create.ACTION=Create Account
admin.PSMerchantAccount.delete.ACTION>Delete Selected Accounts...
admin.PSMerchantAccount.update.ACTION=Update
admin.PSMerchantAccount.cancel.ACTION=Cancel
```

In-line example:

- PSPL file:

```
<screen id="brand" updateID="BRAND" configures="PSMerchantAccount">
  ...
  ...
  <action id="add" safe="1">Add an Account...</action>
  <action id="create">Create Account</action>
  <action id="delete">Delete Selected Accounts...</action>
  <action id="update">Update</action>
  <action id="abort" updateID="$CANCEL" sendNoData="1">Cancel</action>
</screen>
```

PSPL - message element

All error codes and messages are defined in the PSPL files by specifying a *'message'* element inside the PSPL, screen, fieldGroup or field elements. These messages are displayed just above the details information as shown in Figure 12.

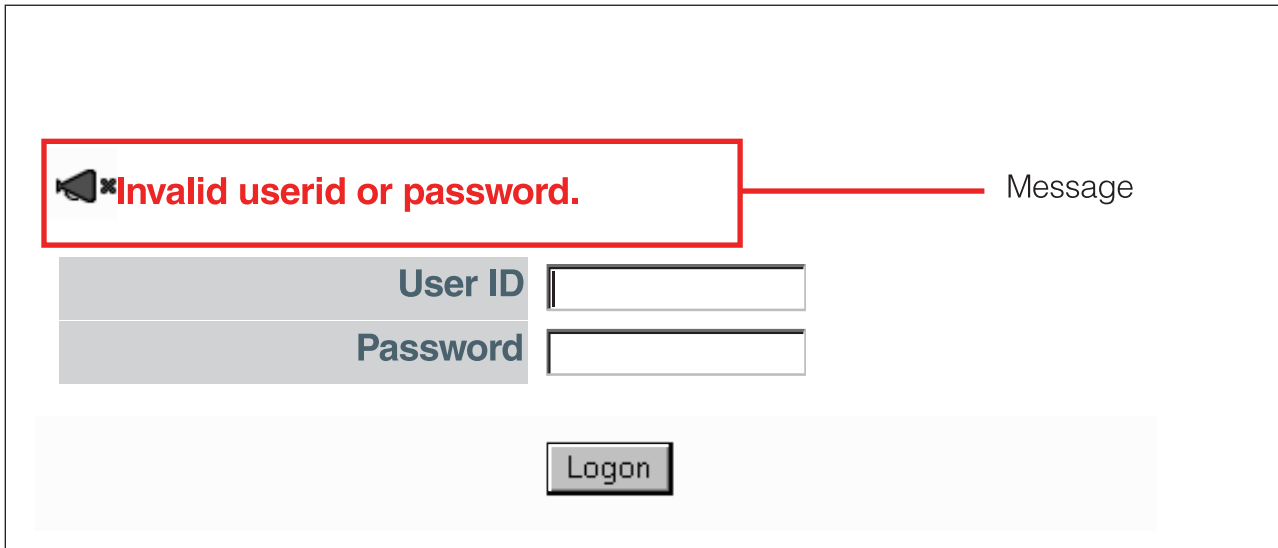


Figure 12. Message element

The *'message'* element attributes are:

Name	Description
id	Identifies the message. This is the name known by the cassette. There are two forms of the id: <ul style="list-style-type: none">• aaaaa- where aaaaa is any value understood by the cassette.• PRC=xx-SRC=yyy where xx is the primary return code and yy is the secondary return code associated with the command.
type	Identifies the type of message. This attribute determines the color of the message and its associated image. Valid values are "warning", "error", or "info".
textId	Key to be used to lookup the message in the associated properties file. Only valid for properties based PSPL.

Properties based example:

- PSPL file:

```
<screen id="OfflineCardAccount" extends="PSMerchantAccount">
  ...
  <fieldGroup id="OfflineCardAccountDetails" advanced="0">
    ...
    <field id="Currency" ...>
      ...
    </field>
    <field id="BatchCloseTime" ... >
      ...
    </field>
  </fieldGroup>
  ...
  <message id="PRC=3-SRC=1060" type="error"
    textid="OfflineCard.OfflineCardAccount.PRC3-SRC1060.MESSAGE"/>
</screen>
```

```

        <message id="PRC=3-SRC=1096" type="error"
            textid="OfflineCard.OfflineCardAccount.PRC3-SRC1096.MESSAGE"/>
        ...
    </screen>

```

- Properties file:

```

...
OfflineCard.OfflineCardAccount.PRC3-SRC1060.MESSAGE=
Error: A 3 digit ISO Currency Code is required but was not specified.
OfflineCard.OfflineCardAccount.PRC3-SRC1092.MESSAGE=
Error: A password is required but was not specified. ...

```

In-line example:

- PSPL file:

```

<screen id="OfflineCardAccount" extends="PSMerchantAccount">
    ...
    ...
    <message id="PRC=3-SRC=1060" type="error">
        Error: A 3 digit ISO Currency Code is required but was not specified.
    </message>
    <message id="PRC=3-SRC=1096" type="error">
        Error: A password is required but was not specified.
    </message>
    ...
</screen>

```

Debugging user interface problems

The following facilities are available to help you trace or debug problems:

- Tracing:
 - The WebSphere trace service, available through the WebSphere Application Server administrative console, enables you to locate syntax errors in your PSPL files. The UI Servlet (and Payment Servlet) can write trace entries into a trace output file. See “Enabling trace” on page 120 for more information.
- On-screen debug:
 - `wpmui.Debug` parameter: This debug parameter enables you to direct the UI Servlet to display intermediate results, statistics, and error information in red color on the generated HTML pages displayed in your browser.

To enable the on-screen debug facility, do the following:

1. Open the WebSphere Application Server administrative console.
2. Click **Servers > Application Servers** in the console navigation tree. On the Application Server page, click the name of the server whose JVM settings you want to configure (the Payments instance name).
3. On the settings page for the selected application server, click **Process Definition**.
4. On the Process Definition Page, click **Java Virtual Machine**.
5. Select **Custom Properties**.
6. The Property page lists the Java system properties for the Payments instance. Click the `wpmui.Debug` property.
7. Specify the value for the debug parameter: 0 (if you want debug off), or 1 (debug on; be sure to enter 1 with no minus sign). Click **Apply**.
8. Restart WebSphere Commerce Payments.

Once the WebSphere Commerce Payments application server is restarted, trace entries will be written into the log file specified for the trace service.

Note: Prior to WebSphere Commerce Payments Version 5.5, you had to format the trace output to read it. Formatting of trace output is no longer required.

Framework Javadoc

The WebSphere Commerce Payments Cassette Programming Reference is documented in the Javadoc. For more information, see the Javadoc packages file.

Cassette view of framework classes

After seeing the large number of framework classes and interfaces that cassette writers must work with, it is time to sort them in terms of *how* the cassette uses each class or interface. As you will see, there are a relatively small number of these classes and interfaces that you will work with on a regular basis. In fact, you will probably never have to directly reference the majority of the framework classes!

Classes and interfaces can be sorted into these categories:

- “Classes cassettes extend”: used to develop your cassette’s infrastructure.
- “Interfaces cassettes implement”: used to develop your cassette’s financial data model and implement the majority of the cassette’s payment processing logic.
- “Classes that provide framework services” on page 155: used to define the service methods that the cassette can use to do its work.
- “Classes on which cassettes operate” on page 155: used to identify the cassette data objects for payment processing.

Classes cassettes extend

Use these classes to develop your cassette’s infrastructure:

- `com.ibm.etill.framework.admin.AdminObject` - you will extend this only if you need to define your own primary administrative objects.
- `com.ibm.etill.framework.cassette.Cassette` - every cassette must extend this class. Here is where you build your cassette infrastructure for the WebSphere Commerce Payments.
- `com.ibm.etill.framework.cassette.query.CassetteQuery` - you will extend this class if you augment query API command results with cassette-specific data. Here is where you build your cassette infrastructure for the Payment Servlet.

For more information on these interfaces, see Chapter 4, “Writing your cassette”, on page 171.

Interfaces cassettes implement

Cassette’s key financial objects are defined as interfaces to provide flexibility to accommodate the needs of different payment protocols. Each cassette writer must decide not only which objects their payment protocol supports, but also whether there is a one-to-one or many-to-one relationship between the order and the other various objects. In one-to-one cases, a single object may implement more than one of these interfaces.

- `com.ibm.etill.framework.cassette.CassetteBatch` - you will implement this interface if your cassette supports batches. This is the Framework’s view of a cassette-specific batch object.
- `com.ibm.etill.framework.cassette.CassetteOrder` - every cassette must implement interface. This is the framework’s view of a cassette-specific order object.
- `com.ibm.etill.framework.cassette.CassetteTransaction` - every cassette must implement interface to handle payment-related events. Additionally, if your

cassette supports refunds, you will also have to implement this interface to handle credit-related events. This is the framework's view of a cassette-specific payment and credit object.

- `com.ibm.etill.framework.io.ComPoint` - if your cassette must support inbound protocol messages, such as messages from a wallet, then you must implement a `ComPoint` object to handle those messages.
- `com.ibm.etill.framework.io.ETillConnection` - depending upon your payment protocol and the communication methods on which it is based, you may need to implement your own `ETillConnection` class. Note that if your cassette only uses basic sockets-based messaging, you may be able to use existing framework-supplied `ETillConnection` classes instead of writing your own.

For more information on these interfaces, see Chapter 4, "Writing your cassette", on page 171.

Classes that provide framework services

Use these classes to define the service methods used by your cassette. Note that both Java classes and Java interfaces are included in this list:

- Core services (heavily used):
 - class `com.ibm.etill.framework.archive.CommitPoint`
 - class `com.ibm.etill.framework.archive.ETillArchive`
 - class `com.ibm.etill.framework.log.ErrorLog`
 - class `com.ibm.etill.framework.log.Trace`
 - class `com.ibm.etill.framework.supervisor.Supervisor`
- Framework communication support (optional):
 - class `com.ibm.etill.framework.io.FrameworkDataStream`
 - class `com.ibm.etill.framework.io.FrameworkServerComPoint`
 - interface `com.ibm.etill.framework.io.HTTPConst`
 - class `com.ibm.etill.framework.io.HTTPInputStream`
 - class `com.ibm.etill.framework.io.HTTPOutputStream`
 - interface `com.ibm.etill.framework.io.MimeConst`
 - class `com.ibm.etill.framework.io.MimeInputStream`
 - class `com.ibm.etill.framework.io.MimeOutputStream`
 - class `com.ibm.etill.framework.io.ServerSocketComPoint`
 - class `com.ibm.etill.framework.io.SocketComPoint`
 - class `com.ibm.etill.framework.io.SyncSocketComPoint`
 - class `com.ibm.etill.framework.io.TimeOutInputStream`

Classes on which cassettes operate

The remaining framework classes and interfaces represent the short-lived and long-lived objects on which your cassette operates. The (many) classes and interfaces can be grouped into these categories:

- Framework financial objects
- Framework administration objects
- Request and response objects
- Framework keys, identifiers, and other miscellaneous objects

Chapter 3. Designing your cassette

After you understand the WebSphere Commerce Payments framework, you can begin designing your cassette. Read this section and look at the LDBCard cassette. LDBCard is a complete and functional cassette infrastructure upon which most other cassettes can be built. LDBCard is designed specifically for credit card processing and it is especially useful when developing other cassettes in this category.

You can retrieve the LDBCard cassette and its accompanying documentation from the same Web site from which you downloaded this *Cassette Kit Guide*. The LDBCard code is packaged with a detailed "cookbook" that walks you through the process of modifying the LDBCard code to implement your own cassette. For many people, this documentation will provide most of what you need. In other cases, however, LDBCard's simplifying assumptions may not apply to the given payment protocol. Yet other cassettes may need to use framework features that are not used in LDBCard. In these cases, you should still use LDBCard's infrastructure as your starting point, augmenting its design as necessary to suit your needs.

In any case, LDBCard is a complete and fully-functioning cassette upon which you can build your own cassette. The number of extensions and modifications to the LDBCard design will vary according to the needs of your cassette.

Design activities

While there are many approaches to software design, the following steps are essential to building a complete payment cassette design. Although these steps are presented in a sequence, portions of each will occur in parallel:

1. "Name your cassette"
2. "Consider internationalization" on page 158
3. "Design your extensions to the framework object model" on page 160
4. "Map the WebSphere Commerce Payments API to your payment protocol" on page 164
5. "Design your commit points" on page 165
6. "Consider restart implications" on page 165
7. "Design ComPoints and ETillConnection" on page 167
8. "Design financial state transitions" on page 167
9. "Create scenario diagrams" on page 168
10. "Write your cassette documentation" on page 168

Name your cassette

The first thing you will do is choose a name for your cassette (throughout this section, XXX = your cassette name). To do this, it is important to understand where and how this name will be used by the framework. The cassette name is configured in the Cassette profile (ETCASSETTECFG) and will be used to:

- Build the name of the Java class loaded for the cassette - XXXCassette
- Build the package name for the cassette (in conjunction with the configured company name)
- Find the properties file for the cassette - XXX.properties

1. Name your cassette. The naming convention for the cassette (including package name) is:

```
com.co_name.ibmetill.ps_namecassette.PSNameCassette
```

Where:

- **co_name** is in lower case and is taken from the COMPANYPKGNAME field of the ETCASSETTECFG table
- **ps_name** is taken from the PAYMENTSYSTEMNAME field of the ETCASSETTECFG table and converted to lower case
- **PSName** is taken from the PAYMENTSYSTEMNAME field of the ETCASSETTECFG table

For example, a valid cassette name would be:

```
com.acme.ibmetill.acmecardcassette.ACMECardCassette
```

IBM-developed cassettes use this convention:

```
com.ibm.etill.ps_namecassette.PSNameCassette
```

The Cassette name should be descriptive, because this is how your cassette will be known to the public. For example:

- VisaNetCassette
- AcmeCardCassette

Consider internationalization

Internationalization is the process of designing an application so that it can be adapted to various languages and regions without having to make code changes. An internationalized program has the following characteristics:

- With the addition of localization data, the same executable can run worldwide.
- Textual elements, such as status messages and the GUI component labels, are not hardcoded in the program. Instead they are stored outside the source code and retrieved dynamically.
- Support for new languages can be easily added and does not require recompilation.
- Culture-dependent data, such as dates and currencies, appear in formats that conform to the end user's region and language.
- It can be localized quickly.

Error messages

Error messages are logged in the `activity.log` file for the cassette. For each language the cassette developer wishes to support, a properties file should exist with the following name: `cassetteName_lang_country.properties`, where `cassetteName` is the name of your cassette, `lang` is the language, and `country` is the optional country if there are different resources for the same country. During Cassette initialization, a `java.util.ResourceBundle` object for the cassette is created by `com.ibm.etill.framework.log.ErrorLog`. `ResourceBundle` objects know how to obtain locale-specific resources. When the cassette attempts to log an error message using the `com.ibm.etill.framework.log.ErrorLog` object, the `ErrorLog` object will find the cassette's `ResourceBundle`, which in turn uses the current locale to determine which properties file to use.

User interface

In WebSphere Commerce Payments, text that is displayable through the user interface as well as and that text's associated attributes are specified in a properties

file that is separate from the PSPL file. A user interface properties file should exist for each language you intend to support in the UI. The properties file should be named as follows:

- `cassetteNameUI_lang_country.properties`, where ***cassetteName*** is the name of your cassette, *lang* is the language, and *country* is the optional country if there are different resources for the same country.

The following lists the languages that are supported by the WebSphere Commerce Payments framework. If you want your cassette to support a specified language, you should create the appropriate cassette properties file. For more information about creating a cassette UI properties file, see “Create your cassette’s UI properties file” on page 131.

Table 11. Languages supported by WebSphere Commerce Payments

Language	Error message properties file name	UI properties file locale	Locale
Brazilian Portuguese	<code>cassetteName_pt.properties</code>	<code>cassetteNameUI_pt.properties</code>	pt_BR
English	<code>cassetteName_en.properties</code>	<code>cassetteNameUI_en.properties</code>	en_US
French	<code>cassetteName_fr.properties</code>	<code>cassetteNameUI_fr.properties</code>	fr_FR
German	<code>cassetteName_de.properties</code>	<code>cassetteNameUI_de.properties</code>	de_DE
Italian	<code>cassetteName_it.properties</code>	<code>cassetteNameUI_it.properties</code>	it_IT
Japanese	<code>cassetteName_ja.properties</code>	<code>cassetteNameUI_ja.properties</code>	Ja_JP
Korean	<code>cassetteName_ko.properties</code>	<code>cassetteNameUI_ko.properties</code>	ko_KR
Simplified Chinese	<code>cassetteName_zh_CN.properties</code>	<code>cassetteNameUI_zh_CN.properties</code>	zh_CN
Spanish	<code>cassetteName_es.properties</code>	<code>cassetteNameUI_es.properties</code>	es_ES
Traditional Chinese	<code>cassetteName_zh_TW.properties</code>	<code>cassetteNameUI_zh_TW.properties</code>	Zh_TW

Data types: The cassette developer should determine if there is any cassette-specific data that can potentially hold translated data. One instance in which this could be the case is for cassette protocol data that comes from the Buyer. For example, if a cassette expects protocol data on the AcceptPayment API command that contains the Buyer’s name, then the `java.lang.String` object in the cassette that represents that data should support the appropriate encoding (such as UTF8) so that the Buyer’s name can be specified in any language supported by the cassette.

Database tables: All translatable data that is stored in the cassette-specific database tables should be defined as VARCHAR FOR BIT DATA. For example, the following cassette database table supports translatable data for the Buyer’s name and address (location) data:

```

SampleCassetteOrder
  MerchantNumber      VARCHAR(9) NOT NULL,
  OrderNumber         VARCHAR(9) NOT NULL,
  AccountNumber       VARCHAR(9) NOT NULL,
  Pan                  VARCHAR(64) NOT NULL,
  Expiry              VARCHAR(64) NOT NULL,
  Brand                VARCHAR(40) NOT NULL,
  StreetAddress        VARCHAR(128) FOR BIT DATA,
  City                 VARCHAR(50) FOR BIT DATA,
  StateProvince        VARCHAR(50) FOR BIT DATA,
  Country              VARCHAR(50) FOR BIT DATA,
  PostalCode           VARCHAR(14) FOR BIT DATA,
  CardHolderName       VARCHAR(64) FOR BIT DATA,
  PRIMARY KEY         (MerchantNumber, OrderNumber)

```

In general, the only translated data that should be stored in the cassette-specific database tables is data that has been supplied as values on protocol data parameters. Do not store translated strings if they can be represented as non-string data, for example, as an enumerated type. In such cases, store the enumerated value and then associate it with a displayable string in a properties file upon display through the user interface. For example, if your cassette uses a parameter called \$COLOR and the possible values are red and green then use the value 1 to represent red and 2 to represent green, and store this value in your database table. Then when displaying this value through the UI, set up your PSPL file to associate the value 1 and 2 with a locale-appropriate strings meaning "red" and "green" respectively in the UI properties file.

Design your extensions to the framework object model

The next step is to design your cassette's persistent data (that is, data that your cassette will store in the database). Aspects to this activity include:

- Identifying all of the cassette's extensions to the framework's object model
- Designing the external view of your extensions (that is, decide which of your extensions should be exposed to applications and users)
- Designing the database tables and views required to support your extensions

Identify all of your cassette's extensions

To begin this activity, you must identify all of the protocol-specific data items that your cassette will need to save as extensions to the framework's objects. Remember that two of the primary divisions of the framework object model are:

- The administrative model, which maintains all of the configuration information upon which your financial transactions will be based
- The financial model, which represents the financial data that comprise your transactions

Cassettes should extend framework administrative objects when they have protocol-specific data to attach to those objects. A common example is the attachment of some protocol-specific information to the AccountAdmin object (typically through the cassette's Account object) that will tell the cassette how to forward financial transactions to financial processors and which processor to use for each given account. The mapping of cassette-specific configuration and administration data to the framework objects should be based upon the scope of the data and the relationship the data represents. To review the role of each administrative object within WebSphere Commerce Payments, see "Framework object model" on page 7.

Cassettes may also need to define their own primary administrative objects. These are implemented as MerchantCassetteObjects or SystemCassetteObjects. An example of such an object is the MerchantCassetteObject named Brand, which the former IBM Cassette for SET supported. Brand is a central concept within the SET protocol; there is no framework object that represents this concept.

Cassettes must also define classes that extend the framework financial object model. The payment protocol supported by the cassette will ultimately determine which of the financial objects need to be reflected in the cassette's model, but **every** cassette must support the Order and Payment objects. Credit and Batch objects are optional, but you are strongly encouraged to present some form of Batches *even if the payment protocol does not have such a concept* to facilitate easy management of daily (or other periodic) Payment and Credit totals for merchants.

More often than not, credit/debit card-based cassettes tend to have one class that correspond to each of the framework financial classes (that is, Order, Payment, Credit and Batch). This method is recommended and is illustrated in the LDBCard cassette.

Design the external view of your extensions

After identifying all of the cassette-specific extensions to the various framework objects and any required cassette-specific primary administration objects (that is, MerchantCassetteObjects or SystemCassetteObjects), you must carefully decide which parts of these should be exposed to application programs and users through the Query commands and WebSphere Commerce Payments user interface. Remember that not all of your cassette-specific data should be exposed. Essentially, if the merchant software or end user requires access to a particular cassette-specific data item to adequately perform their duties, that item should be exposed. If access is not required, the data item should probably remain internalized within the cassette.

One important point about exposing MerchantCassetteObjects and SystemCassetteObjects: Since the Query commands always operate upon framework objects, the cassette's primary administration objects must be exposed as an attachment (called a CassetteConfigObject) to some framework administration object with which the cassette object is somehow associated. Using the example of the former SET Brand objects, each SET Brand was associated with a given Account. Therefore, SET Brands were exposed as CassetteConfigObjects through the results of the QueryAccounts command.

Design your database tables and views

When designing your cassette's database tables and views, remember the key points discussed here, which are also illustrated in `setupTestTables.txt`, found in the cassette kit install directory. As a sample database setup script, `setupTestTables.txt` shows how to set up a cassette's database tables and views; not only for cassette extension objects, but also for a MerchantCassetteObject named Brand. Each of the cassette's extensions contain four to five sample fields, each of a different datatype. All of these fields are made available through the VIEWS for exposure through the Query API set.

- To ensure compatibility with all of the database products supported by WebSphere Commerce Payments, limit the length of your cassette's table and view names to 18 characters or less.
- Typically, a cassette maintains one table for each type of framework object that it extends. Use this naming scheme for your cassette tables (where "xxx" represents your cassette's PaymentSystemName in the ETCASSETTECFG table):

Extensions to...	Table name	Primary Key
-----	-----	-----
Order	xxxOrder	MerchantNumber, OrderNumber
Batch	xxxBatch	MerchantNumber, BatchNumber
Payment	xxxPayment	MerchantNumber, OrderNumber, PaymentNumber
Credit	xxxCredit	MerchantNumber, OrderNumber, CreditNumber
AccountAdmin	xxxAccount	MerchantNumber, AccountNumber
PaySystemAdmin	xxxPaySys	MerchantNumber
CassetteAdmin	xxxCfg	no recommendation (typically contains only one record)

- The column names of the primary key fields in your cassette's tables should match the names of those columns in the corresponding framework tables. This becomes important in building the VIEWS required by the query infrastructure.
- Tables that represent your MerchantCassetteObjects and SystemCassette objects should have keys that will allow you to do a single SQL QUERY that will

retrieve all of the records related to a given framework object. In addition, each of these tables must support the following columns to support the framework's administration infrastructure:

Column Name	Datatype
-----	-----
Enabled	SMALLINT
Active	SMALLINT
Valid	SMALLINT
Pending	SMALLINT
MessagesKey	VARCHAR(40)

- For each cassette-specific extension that will be at all visible through the Query API set, you should provide an SQL VIEW definition that joins the framework table with the associated cassette table. This allows the cassette to use the methods built into the `com.ibm.etill.framework.xdm.QueryRequest` class to execute the SQL QUERY necessary to retrieve the set of cassette objects related to a given framework object.

The following view would be used in the xxx cassette's QueryOrders processing logic:

```
CREATE VIEW xxxOrderView
AS SELECT ETORDERVIEW31.*,
        xxxOrder.Field1 as xxxOrderField1,
        xxxOrder.Field2 as xxxOrderField2,
        xxxOrder.Field3 as xxxOrderField3,
FROM xxxOrder, ETORDERVIEW31
WHERE ETORDERVIEW31.MerchantName = xxxOrder.MerchantNumber
AND ETORDERVIEW31.OrderNumber = xxxOrder.OrderNumber ;
```

By issuing an SQL QUERY against this view, the cassette can extract all of its `CassetteOrder` objects that are associated with the requested framework `Order` objects. Each of its own field names will be accessible by the names assigned in the `SELECT` clause. This assignment is made to avoid any potential name collisions between the column names in the framework's table and those in the cassette's table. Additionally, the framework method in the `QueryRequest` class will be able to access any of the framework fields that it requires (the `ETORDERVIEW31.*` in the `SELECT` clause is required for the `QueryRequest` method to work correctly).

Note: Prior to WebSphere Commerce Payments version 3.1, the `ETORDERVIEW31` view was called `ETORDERVIEW`. If you are migrating a cassette to this release of WebSphere Commerce Payments, please note the migration considerations described in "Migration considerations and steps" on page 182.

For more information, examine the remaining views in `setupTestTables.txt`. In particular, the `Payment` and `Credit` extension objects are defined with two different `VIEWS`:

- One `VIEW` for `QueryPayments` and `QueryCredits` commands
- The other `VIEW` for `QueryBatches` commands that request payment and credit detail.

For more examples, see `setupTestTables.txt`, which is in the cassette kit install directory.

- Consider adding growth fields to each of your cassette tables to allow table additions during the life of your cassette. Once your cassette is in production, it is usually not possible to make changes to your database tables. The growth fields allow changes to be made for things such as bug fixes and functional

enhancements. In addition, using growth fields for table additions needed for a new release of your cassette will minimize future database table migration efforts. Suggested growth fields are:

Growth1	INTEGER
Growth2	INTEGER
Growth3	VARCHAR(150) FOR BIT DATA
PersistFdsKey	VARCHAR(40)

The Growth fields should be used as efficiently as possible. For example, the Growth3 field could hold several 20 character pieces of data since it is 150 characters in length. When storing multiple pieces of data into one field, you must have some sort of index in the field that indicates which pieces of data are present and their length. The PersistFdsKey (persistent fields key) should be used as a key into the ETBINARYDATA table.

- Datatype guidelines:

When defining fields to reflect or match fields in any of the framework tables, use the same datatypes as the framework table definitions. Some of the more frequently-used fields are:

Column Name/content	Datatype
MerchantNumber	VARCHAR(9)
OrderNumber	VARCHAR(9)
BatchNumber	VARCHAR(9)
PaymentNumber	VARCHAR(9)
CreditNumber	VARCHAR(9)
AccountNumber	VARCHAR(9)
Amount	INTEGER
ReferenceNumber	VARCHAR(64)
CardholderID	VARCHAR(64)
any time stamp	TIMESTAMP
any boolean	SMALLINT
state variables	INTEGER
keys into ETBINARYDATA table	VARCHAR(40)
IP hostnames	VARCHAR(256)
TCP, UDP port values	INTEGER
strings that will potentially contain international characters	VARCHAR FOR BIT DATA(length as needed)
any other strings	VARCHAR(length as needed)

Define commonly-used cassette-specific fields consistently across cassettes.

Recommended types for common credit/debit card-specific fields include:

Column Name	Datatype
Brand	VARCHAR(40)
PAN	VARCHAR(64)
Expiry	VARCHAR(64)
CardVerifyCode	VARCHAR(4)
AVSStreetAddr	VARCHAR(128)
AVSCity	VARCHAR(50)
AVSStateProvince	VARCHAR(50)
AVSCountry	VARCHAR(50)
AVSPostalCode	VARCHAR(14)
CardHolderName	VARCHAR(64)

- Consider if any of the data in the database should be encrypted.

For security reasons, sensitive data, such as credit card numbers and checking account numbers should not be stored in the database in the clear. The framework does not provide encryption/decryption services. Encryption and decryption support must be provided by the cassette developer.

Map the WebSphere Commerce Payments API to your payment protocol

Mapping your payment protocol to the WebSphere Commerce Payments API requires a thorough understanding of both. Once you understand the central data in your payment system and how it relates to the framework's data, you are ready to design this mapping.

Mapping your payment protocol to the WebSphere Commerce Payments API means defining which actions occur within your cassette in response to each API command received from the merchant or administrator. To do this, you also need to understand what actions will occur within your cassette in response to your own payment protocol messages received from the outside world. Mapping your payment protocol to the generic payment API means defining which actions occur within your payment system in response to each API command received from the merchant. To do this, you also need to understand what actions will occur within your payment system in response to your own payment protocol messages received from the outside world.

During this mapping process (and, for that matter, during the entire cassette development process), remember that the user of the cassette is merchant software, and that the merchant software views your payment protocol through the WebSphere Commerce Payments API. Therefore, it is important to create this mapping from the viewpoint of the WebSphere Commerce Payments API rather than trying to figure out how to expose each and every feature of the payment protocol through the API. For example, ask the question "what should happen if a `DepositReversal` call is made to my cassette?" instead of "how can I expose my protocol's `voidAndRepurchase` function into the generic API?." For more suggestions and conventions for cross-cassette compatibility, see "Cassettes and command processing" on page 31.

Start your API mapping by deciding how Orders will be first created for your cassette. There are two choices, via the `ReceivePayment` API command or via the `AcceptPayment` API command. To understand the distinction and choose which command your cassette will respond to, study the "Framework commands" on page 24. It is legitimate to respond to both, if they both make sense in your payment system.

Next decide how the `autoApprove` and `autoDeposit` flags on the `Receive` and `AcceptPayment` commands will be handled in your cassette:

- Are these flags used?
- Is one or both flags required to be true for all calls to your cassette?

Remember, `autoApprove` should behave as an `approve` in terms of the framework and cassette specific objects which must be created. `AutoApprove` should cause a payment to be created and change to an appropriate post-Approve state - see "Financial objects and their states" on page 11. `AutoDeposit` should cause the payment to change to an appropriate post-Deposit state.

Implementation of the rest of the API commands is optional. The cassette can return a "command not supported" code (`PRC_CASSETTE_ERROR`, `RC_COMMAND_NOT_SUPPORTED`) for any API command which does not make sense for your payment protocol. It is critical to consider the effect of each API command on the state and existence of framework data model objects. The mapping of command to cassette function depends on both a) the mapping of the data object state changes and b) the name of the API command.

Design your commit points

The cassette is responsible for committing all financial data (for example, Order and Payment), including framework data, to the database. The framework will create and update the framework financial objects in memory, using framework objects' methods which do the updates. However, committing the framework financial objects to the database must be coordinated with the commitment of cassette-specific financial objects to the database to prevent inconsistent data in the database due to errors occurring between the commit of the framework financial objects and the commit of the cassette objects. This means that the cassette is responsible for actually invoking the commit operations for Payment commands.

In general, all related financial objects should be committed to the database when their state is consistent and a potentially long wait period is about to be entered. This will ensure that the objects' states are preserved in case of a cassette or system shutdown and will allow the cassette to resume the operation from the point where it left off when the cassette is eventually restarted.

For administration data, the cassette is responsible for committing all Merchant Cassette objects and System Cassette objects, to the database in response to Create, Modify, or DeleteMerchantCassetteObject commands and Create, Modify, or Delete or DeleteSystemCassetteObject commands. In these cases, there are no corresponding framework objects to be committed to the database. For Cassette Extension objects, such as a cassette-specific extension of the framework's AccountAdmin object, the framework will commit the changes to the database. In these cases, the cassette is responsible for adding new, updated, or deleted cassette extension objects to the appropriate commit point list, but the framework will invoke the commit operations when command processing is complete. An example of a command in this category is the CreateAccount command.

The LDBCARD cassette includes well-defined commit points. Not only does it show how to use the Archivable interface and the thread's CommitPoint object, but it also shows you *when* objects should be committed.

Consider restart implications

Before you go any farther, you should think about what type of restart capabilities you will want to build into your cassette:

- For example, if your cassette must communicate with some sort of "back end" system (an acquiring institution, for example), then how will your cassette handle the case where you send a request to the back end, and then the cassette is stopped (either intentionally or unintentionally)?
- Will you save the state of the back-end request so that it can be restarted once the cassette is restarted, or should you simply not update any database entries until the back-end response is received?
- What will the back-end system do if you reissue a request that it has already processed? Does that system maintain idempotent semantics?
- What data needs to be saved to restart a back-end operation?
- What type of state information should be saved with your financial objects before sending back-end requests?
- How will you handle new requests against the objects for which the pending operation was issued? Should any new states be included in your FSM to account for this?

As you can see, the answers to these questions can significantly affect your cassette's overall design. While we cannot offer concrete answers to these questions, we can offer some important suggestions to make your overall cassette design more elegant, robust and maintainable:

- Most cassettes will have to handle communication failures and situations created by the fact that the account, payment system, cassette, or WebSphere Commerce Payments have been stopped or shut down while waiting for a response message from the back end. If your cassette is one of these, it is imperative that you build your cassette architecture and design to handle restart scenarios. Do not wait until you have a working cassette to "fit in" restart logic later. Because of the different commit points and potential points of failure involved in such a system, it quickly becomes very difficult to force this type of restart recovery over an existing architecture.
- You must clearly understand your payment protocol's capabilities concerning the handling of duplicate messages. If the protocol is idempotent, then you can safely issue duplicate messages to the back end without fear of causing an action to occur more than one time. If it is not, you may need to use other capabilities of the protocol (for example, queries to establish the state of a transaction at the back end) before reissuing messages.
- Maintain a clear separation between the generation of outbound messages, the processing of inbound messages, and the actual act of sending or receiving those messages. This is an integral part of an architecture that allows for restart scenarios.
- Try to limit the amount of time it will take for a given API request to complete. Remember that there are a fixed number of API threads and service threads in the Payment Servlet, so every time you send a message and wait for a response, you are suspending one of those threads and making it unavailable to other users of WebSphere Commerce Payments. Therefore, you should try to limit the amount of time that any of these thread will spend waiting for a response message. If the response time limit is reached, then build a `com.ibm.etill.framework.cassette.CassetteWorkItem` for the request, add it to the framework's timer queue using the `com.ibm.etill.framework.supervisor.Supervisor.addItemToTimerQueue` method to process the work item after some period of time, and then throw an `ETillAbortOperation` with `PRC_OPERATION_PENDING` and a secondary return code that indicates which framework object is being operated upon (for example, `RC_ORDER` or `RC_PAYMENT`).

To aid the cassette developer with restart, WebSphere Commerce Payments request objects (that is, the `com.ibm.etill.framework.cassette.CassetteRequest` object hierarchy) are serializable, so the request can be serialized and saved to the database. It is important to note that all WebSphere Commerce Payments classes that implement the `java.io.Serializable` interface contain their own definition of a serial version ID. A serial version ID is a secure hash of the full class name, superinterfaces, and members -- the facts about the class that, if they change, signal a possible class incompatibility. Defining our own serial version ID is needed to protect us in the situation where a serializable class's implementation changes between the time an object is serialized and stored in the database and the time the object is retrieved from the database and deserialized. Since each class defines its own serial version ID, when the `ObjectInputStream` attempts to deserialize an object and compares the actual serial version ID with the serial version ID defined in the class, the IDs will be the same even though the implementation may have changed. If you find a need to create your own serializable objects for storage to and instantiation from the database, you need to do define a serial version ID for the class as well. Obtaining serial version IDs is accomplished by using the `java`

command serialver. Once you have the serial version ID, define it in your class. Here's an example of how to do that from the `com.ibm.etill.framework.cassette.AcceptPaymentRequest` class:

```
public class AcceptPaymentRequest extends OrderRequest
{
    static final long serialVersionUID = -3273221806051804609L;
    .
    .
    .
}
```

Design ComPoints and ETillConnection

Many cassettes will expect to receive inbound protocol messages that are not from the merchant. For example, any protocol that relies on a wallet application at the consumer workstation will receive such messages. Because the messages are protocol-specific, only the cassette knows their form and content.

The framework will provide threads dedicated to listening for these incoming messages on your cassette's behalf. For each "place" messages may arrive, the cassette must provide an object of a type derived from `ComPoint` which waits for notification of an incoming message. This object must return an object of a type derived from `ETillConnection` that represents the connection (or pipe) to the source of the incoming message. The `ETillConnection` object will be passed to the cassette as it is asked to handle the incoming message. The `ComPoint` object will return to waiting for the next incoming message.

To view the information in the JavaDoc, see:

- `com.ibm.etill.framework.io.ETillConnection`
- `com.ibm.etill.framework.io.ComPoint`

Design financial state transitions

The goal of this step is to understand the framework object states and then design how your cassette will perform the transitions between those states. To achieve this design, you must understand, in a fair amount of detail, what your cassette will do in response to each type of incoming financial request. The financial requests your cassette will have to handle include, but are not limited to:

- The full set of payment commands (administrative and query commands are not included here)
- All protocol messages (if any)
- Any financial events that the cassette schedules internally. These events are typically queued onto the framework's timer queue. When the event is eventually generated, the associated requests are driven through the cassette's service method on one of the framework's service threads.
- All events that might occur during a restart of your cassette. This typically involves the resumption of operations that were in progress when the cassette was shut down. Because the cassette is responsible for resurrecting each of these pending operations, the exact nature of the event is up to the discretion of the cassette writer. Whatever that event's nature, though, each such event must be handled by a cassette FSM.

For each of these requests, you must decide:

- What actions should occur in your payment cassette?

- How should the completion of each request be reflected in the framework financial object states?

While the cassette may not define its own states for the framework financial objects, it may define its own intermediate states for its own objects as necessary. This is typically required when the `receivePayment` command is supported (to track the exchange of protocol messages to complete the order creation phase) or when more than one protocol message is required between the cassette and the protocol's "back end" to complete a single API command. In such cases, the cassette should maintain the affected framework financial objects in a "pending" state while its own objects' intermediate states change as a result of processing a sequence of protocol messages.

A key point to understand here is that merchant software should **never** be required to understand any object states other than those defined by the framework. The use of intermediate cassette-specific states should at best be completely internalized within the cassette and at worst, be reflected as informational extensions through the query command output.

Create scenario diagrams

Scenario diagrams show the processing that results from the receipt of a request. By creating a scenario diagram for every request that your cassette will handle, you ensure that you have considered all the basic issues. The scenario diagram will pull together your design work on the object model, API mapping, ComPoint design, and state transitions.

To create your scenario diagrams:

1. Start with the framework diagrams, shown in Chapter 2, "Understanding the WebSphere Commerce Payments framework", on page 7.
2. Add your major Cassette objects across the top of the diagram.
3. Expand the flows to your cassette to show how the processing is handled within your cassette.

Write your cassette documentation

Document your own cassette as a guide for the merchants who will be using it:

- Describe the general concepts behind your cassette and its payment protocol. That is, what is the "big picture" when your cassette is involved?
- Document your object model, including the attribute names by which your data elements are exposed through the query API commands. Also include a discussion on how the cassette's object model "fits into" the framework object model. For example, what does the Account object represent in cassette terms? What is the relationship between a framework Batch object and the batches in the cassette's world? This section will serve as a foundation for the remainder of your documentation.

For an example of cassette object documentation, see the *WebSphere Commerce Payments Cassette for VisaNet Supplement* .

- Document the installation and configuration requirements and procedures. You are encouraged to use the tutorial approach that is used in the WebSphere Commerce Payments cassette supplements to be consistent with documentation for other cassettes.
- *Thoroughly* document how *every* WebSphere Commerce Payments API command is supported by your cassette. Include information such as:
 - Whether your cassette supports the command or not

- If the command behaves differently in different situations, describe the various behaviors. For example, "DEPOSIT causes Payment objects to enter PAYMENT_DEPOSITED state in situation X and PAYMENT_CLOSED state in situation Y".
- If it is not obvious or "standard," describe how the various framework command parameters are treated by your cassette.
- Fully describe any protocol data parameters, including the keywords, the parameter value formats, whether the parameter is optional or required, and how the cassette treats or uses the parameter data.
- Describe *all* of the return codes that your cassette will generate. If there are any cassette-specific return codes (that is, secondary codes with a value greater than 10000), be sure to precisely define the conditions that will cause the error to occur.
- For Query commands, describe in detail the XML representation of all of the cassette-specific extensions to the framework objects (that is, each `CassetteExtension` element) and all of the cassette's `MerchantCassetteObjects` and `SystemCassetteObjects` (represented as `CassetteConfigObject` elements). For each attribute (`CassetteProperty` element) within each of these elements, provide:
 - The attribute name (`propertyId`)
 - What the attribute represents
 - A description of the data type, if appropriate
 - An example of the XML representation of such an object
- Explain error messages and actions that can be taken to resolve errors.
- Provide a tutorial to guide new users through the setup and use of your cassette. You should also provide a mechanism to create the orders that you will use in this tutorial. WebSphere Commerce Payments provides a Sample Checkout application, which is a simple order entry system that supports multiple payment methods. Each payment method is supported by a Cashier profile, which is an XML document that describes how orders should be created for that payment method. See section "Create Cashier profiles (optional)" on page 196 for details on how to create a Cashier profile for your cassette. In addition, see the *WebSphere Commerce Payments Programming Guide and Reference* for details on the Sample Checkout application. Providing a tutorial may require that you provide access to a common test account (if such a thing is available) to which test transactions can be sent.

Chapter 4. Writing your cassette

After you have completed the cassette design activities, you are ready to get down to the business of writing the code.

You are strongly encouraged to use the LDBCard cassette as the skeleton upon which you code your cassette. If you are doing this, then you should use the LDBCard "cookbook" for very specific instructions that will guide you through the source code modifications necessary to build your cassette on top of LDBCard.

This chapter serves as a general guideline for the order in which you should implement your cassette source code. Use an iterative process that first builds the cassette in this order:

- Basic infrastructure
- Administration objects, including external views
- Core protocol function (this makes up the internal cassette classes that interact with the financial institution or other "back end" processor)
- Basic order and payment processing
- Batch processing
- All remaining transactional support
- Installation and packaging

This implementation order is chosen because:

- It produces functional, testable code very quickly.
- It allows you to experience the "look and feel" that users will see very early in the development process. This facilitates early validation of your administration model and assumptions.
- It allows for compartmentalized testing of functional components of your cassette, rather than requiring you to write the majority of your code before ever attempting to execute it.
- Experience shows that this order is very effective in building robust and reliable cassettes.

A payment cassette for WebSphere Commerce Payments is similar to a Java servlet. Just as a servlet provides a service to a server framework, a payment cassette provides payment processing services to the WebSphere Commerce Payments framework. In the same way a servlet services a `ServletRequest` using a `ServletResponse`, a payment cassette services a `CassetteRequest` using a `CassetteResponse`.

Within WebSphere Commerce Payments, there are two categories of cassette requests that a payment cassette will be asked to handle. The first category is the administrative and payment API command requests. The second category contains protocol message requests, which are defined by the cassette itself. When WebSphere Commerce Payments receives a protocol message for a payment cassette, it will first ask the cassette to decode the message and create the appropriate protocol message request. Then the cassette will be asked to service the protocol request.

When servicing these requests from the framework, the payment cassette must maintain the state of framework payment objects. For an explanation of these states, see “Financial objects and their states” on page 11. For an understanding of when state updates need to be performed for a “typical” cassette (that is, one whose object model closely reflects that of the framework, see the LDBC card cassette).

There are several mandatory Java classes and interfaces that each cassette must extend or implement. These classes and interfaces are:

- `com.ibm.etill.framework.cassette.CassetteOrder`
- `com.ibm.etill.framework.cassette.CassetteTransaction` (for payments)

Next, there is a set of classes and interfaces that are technically optional, but which must be implemented to build a robust cassette whose interface is consistent with that expected by a WebSphere Commerce Payments application program or user. These are:

- `com.ibm.etill.framework.cassette.CassetteBatch`
- `com.ibm.etill.framework.cassette.query.CassetteQuery`

Finally, depending upon the payment protocol and the complexity of the cassette implementation, you may choose to implement some or all of:

- `com.ibm.etill.framework.cassette.CassetteTransaction`
- `com.ibm.etill.framework.cassette.ProtocolRequest`
- `com.ibm.etill.framework.admin.AdminObject`

The Javadoc documentation for the framework cassette interface classes and the framework Scenario Diagrams are essential resources for understanding how to write your cassette. The Scenario Diagrams will give you the overall picture of the processing that needs to happen for each incoming event/Request. The Javadoc describes in detail the responsibilities and requirements of each method you need to implement.

The rest of this chapter provides a high level look at what you will be doing in the sequence described above. The only exception is that we will discuss installation issues first because you will need much of this information to test your code during your development phase. Specifically, the following topics are described:

1. “Installation and uninstallation considerations and steps” on page 173
2. “Build a working cassette skeleton” on page 186
3. “Build your administration objects” on page 187
4. “Build your external view of administration objects” on page 187
5. “Implement your core protocol function” on page 190
6. “Implement the basic `CassetteOrder`” on page 190
7. “Implement basic cassette payments” on page 192
8. “Implement `CassetteBatch`” on page 193
9. “Complete cassette payment” on page 195
10. “Implement cassette credits” on page 195
11. “Complete the remaining transactional support” on page 196
12. “Understand platform-specific issues” on page 197

Installation and uninstallation considerations and steps

Before we start coding, we will discuss the basic steps required as well as the information that's available to you to successfully install a payment cassette under an existing WebSphere Commerce Payments framework. Unfortunately, there is currently no prepackaged installation procedure skeleton that we can provide to you. Therefore, you must choose and build your own installation and uninstallation procedures.

Your cassette's installation process should:

1. Use the WebSphere Commerce Payments framework installation information.
2. Perform initial checks.
3. Copy the cassette files to the target system. Supported platforms include: AIX, Solaris, Linux, Windows 2000, and iSeries™. z/OS™ installation information is provided upon request.
4. Perform migration if necessary.

Multiple WebSphere Commerce Payments instances are supported using a single code installation, so you should provide separate procedures for installing your cassette (the above steps 1 to 3) and migrating or adding your cassette to an existing WebSphere Commerce Payments instance (step 4).

Cassette configuration beyond this should be performed using the WebSphere Commerce Payments user interface.

Installing and configuring your cassette

WebSphere Commerce Payments Version 5.5 requires WebSphere Application Server Version 5. Under WebSphere Application Server Version 5, the WebSphere Commerce Payments directory structure changed along with the configuration of WebSphere Application Server relative to that of earlier releases of WebSphere Commerce Payments. As a result of these changes, the way you install and configure your cassette under WebSphere Application Server is different than before. This section describes how to install and configure a cassette for a WebSphere Commerce Payments environment.

Step 1: Create the directory structure

On the machine where you have installed WebSphere Commerce Payments Version 5.5, create the following directory structure for your cassette:

```
WC_installdir/payments/cassettes/cassetteName
```

The *WC_installdir*/payments/cassettes directory should already exist on the system.

In the *cassetteName* directory, create this structure:

```
<cassetteName>/cassette_properties.xml
  schema/
    createTables.db2
    dropTables.db2
    Oracle files...
  lib/
    JAR files
  properties/ (optional)
    properties files
  psp1/
    *.pspl
```

```

profiles/          (optional)
  *.profiles
OnlineHelp/       (optional, include locales for desired languages)
  en/*.html
  de/*.html
  ...
Documentation/    (optional, include locales for desired languages)
  en/*.pdf
  de/*.pdf
  ...
Dynamic_Library/ (optional)
  *.a
  *.dll
  *.so
  ...

```

In this directory structure, do the following:

1. In the `lib` directory, supply the JAR file containing the cassette code you have written.
2. In the `pspl` directory, supply at least one PSPL file.
3. In the `schema` directory, supply the database scripts to create and drop database tables for the database you are using (DB2 or Oracle). If you are migrating your cassette, you can also place migration scripts in this directory.
4. Create a file called **cassette_properties.xml** and place it in the `cassetteName` directory. Ensure that the XML file follows the schema shown in the `Cassette.xsd` file in the `WC_installdir/payments/cassettes` directory.

Example of a `cassette_properties.xml` file

An example of a `cassette_properties.xml` file follows. In the example `xxxCassette` represents the name of a cassette. The example also shows the use of markup for optional configuration elements. For example, these are optional configuration elements:

- `Cassette:RequiredParameterInfo`
- `Cassette:SystemPropertyInfo`
- `Cassette:ConfigClass`
- `Cassette:MigrateScript`

```

<?xml version="1.0" encoding="UTF-8" ?>
<Cassette:CassetteProperties xmlns:Cassette="http://www.ibm.com/websphere/commerce/payments" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ibm.com/Cassette.xsd">

<Cassette:CassetteInfo>
  <Cassette:CassetteName>xxxCassette</Cassette:CassetteName>
  <Cassette:CassetteVersion>2.8.0.0</Cassette:CassetteVersion>
  <Cassette:CassetteMinFrameworkVersion>5.5.0.0</Cassette:CassetteMinFrameworkVersion>
  <Cassette:CassetteVendor>MyCompany</Cassette:CassetteVendor>
</Cassette:CassetteInfo>

<Cassette:ConfigInfo>
  <Cassette:ArchiveName>xxxCassetteClasses.jar</Cassette:ArchiveName>
  <Cassette:PSPL>xxxCassette_1.psql</Cassette:PSPL>
  <Cassette:PSPL>xxxCassette_2.psql</Cassette:PSPL>
  <Cassette:ConfigClass>com.myCompany.xxxCassetteConfigurator<Cassette:ConfigClass>
</Cassette:ConfigInfo>

<Cassette:DatabaseInfo>
  <Cassette:DatabaseType>DB2</Cassette:DatabaseType>
  <Cassette:CreateScript>createXxxCassetteTables.db2</Cassette:CreateScript>
  <Cassette>DeleteScript>dropXxxCassetteTables.db2</Cassette>DeleteScript>
  <Cassette:MigrateScript>migrateXxxCassetteTables.db2</Cassette:MigrateScript>
</Cassette:DatabaseInfo>

<Cassette:DatabaseInfo>
  <Cassette:DatabaseType>Oracle</Cassette:DatabaseType>
  <Cassette:CreateScript>createXxxCassetteTables.oracle</Cassette:CreateScript>
  <Cassette>DeleteScript>dropXxxCassetteTables.oracle</Cassette>DeleteScript>
  <Cassette:MigrateScript>migrateXxxCassetteTables.oracle</Cassette:MigrateScript>
</Cassette:DatabaseInfo>

<Cassette:RequiredParameterInfo>
  <Cassette:RequiredParameterID>1</Cassette:RequiredParameterID>
  <Cassette:RequiredParameterName>CassetteEncryptionKey</Cassette:RequiredParameterName>
</Cassette:RequiredParameterInfo>

<Cassette:RequiredParameterInfo>
  <Cassette:RequiredParameterID>2</Cassette:RequiredParameterID>
  <Cassette:RequiredParameterName>CreateSSLTables</Cassette:RequiredParameterName>
</Cassette:RequiredParameterInfo>

<Cassette:SystemPropertyInfo>
  <Cassette:PropertyID>1</Cassette:PropertyID>
  <Cassette:PropertyName>HostName</Cassette:PropertyName>
  <Cassette:PropertyValue>mycomputer.city.company.com</Cassette:PropertyValue>
</Cassette:SystemPropertyInfo>

<Cassette:SystemPropertyInfo>
  <Cassette:PropertyID>2</Cassette:PropertyID>
  <Cassette:PropertyName>PortNumber</Cassette:PropertyName>
  <Cassette:PropertyValue>20</Cassette:PropertyValue>
</Cassette:SystemPropertyInfo>

</Cassette:CassetteProperties>

```

Figure 13. Example of a cassette_properties.xml file. See the Cassette.xsd file for all rules applied to the cassette_properties.xml file. The Cassette.xsd file exists in the WC_install_dir/payments/cassettes directory.

For an explanation of the elements contained in the cassette_properties.xml file, see Table 12 on page 176.

Elements in a `cassette_properties.xml` file

Table 12. Elements in the `cassette_properties.XML` file

Element name	Re-quired element?	Multi-ple elements allowed?	Exists in element	Details
Cassette:CassetteProperties	Yes	No	main section	The main element for this XML document.
Cassette:CassetteInfo	Yes	No	Cassette:CassetteProperties	Section that contains general cassette information.
Cassette:CassetteName	Yes	No	Cassette:CassetteInfo	The name of the cassette.
Cassette:CassetteVersion	Yes	No	Cassette:CassetteInfo	The version of the installed cassette.
Cassette:MinFrameworkVersion	Yes	No	Cassette:CassetteInfo	The minimum version of the WebSphere Commerce Payments component that must be installed for this cassette to run.
Cassette:CassetteVendor	Yes	No	Cassette:CassetteInfo	The name of the company that has written the cassette.
Cassette:ConfigInfo	Yes	No	Cassette:CassetteProperties	Section that contains general cassette configuration information.
Cassette:ArchiveName	Yes	Yes	Cassette:ConfigInfo	The name of the JAR files that are shipped by the cassette which contain the cassette code. Specifically, these are the JAR files that must go into the Payments instance EAR file. Each JAR file should be listed in its own Cassette:ArchiveName element.
Cassette:PSPL	Yes	Yes	Cassette:ConfigInfo	The name of the PSPL files that are shipped by the cassette. Each PSPL file should be listed in its own Cassette:PSPL element.
Cassette:ConfigClass	No	No	Cassette:ConfigInfo	The ConfigClass is the name of a custom extension class that implements the ICassetteConfigurator interface that any cassette can provide. (The ICassetteConfigurator interface is described in the Javadoc in <code>cassette_kit_unzip_dir/docs/javadoc/configuration</code> .)
Cassette:DatabaseInfo	No	Yes	Cassette:CassetteProperties	This section contains information about scripts to manage database tables. Each DatabaseInfo section describes the scripts for one type of database.

Table 12. Elements in the *cassette_properties.XML* file (continued)

Element name	Re-quired element?	Multi-ple elements allowed?	Exists in element	Details
Cassette:DatabaseType	No	No	Cassette:DatabaseInfo	The type of the database to which the scripts apply. The only valid values are DB2 and Oracle.
Cassette:CreateScript	No	No	Cassette:DatabaseInfo	The name of the script used to create database tables.
Cassette>DeleteScript	No	No	Cassette:DatabaseInfo	The name of the script used to delete database tables.
Cassette:MigrateScript	No	No	Cassette:DatabaseInfo	The name of the script used to migrate the database tables.
Cassette:RequiredParameterInfo	No	Yes	Cassette:CassetteProperties	Parameters required for the cassette. In Figure 13 on page 175, you will notice XML elements for required parameters. There are no required parameters for IBM-provided cassettes, nor are they required for any cassette you write. You may have a need for required parameters if you must collect information from your cassette users before the cassette is added to a Payments instance. For example, suppose you needed to gather bootstrap data that must be in the cassette database tables for the cassette servlet to initialize, or you needed to obtain a password for the cassette. You could specify these items as required parameters in your cassette's XML file. The information would then be collected from the administrator before the cassette is added to the instance.
Cassette:RequiredParameterID	No	No	Cassette:RequiredParameterInfo	A unique ID number associated with each RequiredParameterInfo element.
Cassette:RequiredParameterName	No	No	Cassette:RequiredParameterInfo	The name of the parameter needed.

Table 12. Elements in the `cassette_properties.XML` file (continued)

Element name	Re-quired element?	Multi-ple elements allowed?	Exists in element	Details
Cassette:SystemPropertyInfo	No	Yes	Cassette:CassetteProperties	System property information. In Figure 13 on page 175, the use of elements for specifying system properties is not shown. However, as shown in the <code>Cassette.xsd</code> file, you can specify Java system properties in WebSphere for the cassette code to run. The WebSphere Configuration Manager handles the setting of the Java system properties in WebSphere. After properties are set, they are all treated as instance properties. They can be updated through the <code>PaymentsConfigurator.update()</code> method. System properties must use the following notation: <code>cassette.cassetteName.<property name></code>
Cassette:PropertyID	No	No	Cassette:SystemPropertyInfo	A unique ID number associated with each <code>SystemPropertyInfo</code> element.
Cassette:PropertyName	No	No	Cassette:SystemPropertyInfo	The name of the system property.
Cassette:PropertyValue	No	No	Cassette:SystemPropertyInfo	An optional default value for the system property.

“Step 2: Determine if additional configuration is required” walks you through what you should consider before using these elements.

Step 2: Determine if additional configuration is required

Determine if you need to implement the `ICassetteConfigurator` interface

After setting up the directory structure for your cassette, determine if it will need to write a class that implements the `ICassetteConfigurator` interface. Cassettes that you write may need to do more configuration than what the `PaymentsConfigurator` class will handle.

To determine if your cassette needs to write an extension class, consider the following:

- Does your cassette need to configure the database outside of the `createTables` script? If it does, you may want to implement the `add` method to configure the database. A connection to the database is provided (see the `IDatabase` interface description that follows in “Step 2A: (Optional) Use custom extension class if

necessary”). If the cassette implements the add method, the createTables script is not run for the cassette. The cassette is responsible for creating its own tables.

- Does your cassette need any special parameters to be entered when the cassette is added to a Payments instance? If it does, you should implement the add method and list the parameters needed in the `Cassette:RequiredParameterInfo` section of the `cassette_properties.xml` file. (See Table 12 on page 176 for a description of the `RequiredParameterInfo` element in the XML file.) The Configuration Manager will prompt the user for the parameters before the addition, and pass the values collected to the cassette on the add method through the Map object.

Also, as cassette writer, you should ensure that your cassette’s database tables are set up to include any special parameter data.

- Does your cassette need to set any Java system properties in WebSphere for the cassette code to run? If it does, you should implement the `getSystemProperties` method and pass back a Map object that contains the system property names and values to be set for the instance. (See Table 12 on page 176 for a description of the `SystemPropertyInfo` element in the XML file.)

Configuration Manager handling of your cassette’s configuration

The WebSphere Commerce Configuration Manager does the following when your cassette is added or removed:

- It determines if the cassette is installed properly by checking for the existence of the following files:
 - `cassette_properties.xml`
 - JAR file listed in the XML file in the `lib` directory
 - Any PSPL files in the `pspl` directory
- It collects data for any required parameters listed in the XML file.
- It determines if a `ConfigClass` value was provided in the XML file:
 - If no `ConfigClass` was provided, the `PaymentsConfigurator` will create the cassette tables by running the database script for the database type used by the instance.
 - If a `ConfigClass` is provided, the class will be loaded by the `PaymentsConfigurator` and the `add()` method implemented from `ICassetteConfigurator` will be called. If the `add()` method is overwritten, it will be expected to create its own database tables.
- It updates the cassette configuration (`etCassetteCfg`) table with the appropriate information.
- It updates the WebSphere configuration for the instance by adding the cassette’s Java archive file to the `wc.mpf.ear` file, PSPL, and documentation files. It then redeploys the EAR file, if necessary, for the instance to which the cassette is being added.
- It sets or removes any Java system properties required by the cassette in the WebSphere configuration for the Payments instance.

Step 2A: (Optional) Use custom extension class if necessary

The custom extension class allows a cassette writer to configure a cassette beyond what the `PaymentsConfigurator` is capable of doing. This class should exist in one of the JAR files listed in the `Cassette:ArchiveName` elements of the `cassette_properties.xml` file. The class will be loaded by the `PaymentsConfigurator` at run time if the custom class is listed in the `Cassette:ConfigClass` element. To compile your code, ensure that the `WC_installdir/payments/lib/eTillConfig.jar` file is provided in your classpath.

ICassetteConfigurator interface

A custom class must implement the ICassetteConfigurator interface. If the custom class only extends this interface, all the methods must be implemented. The PaymentsConfigurator has provided two classes that fully or partially implement these methods to make writing a custom class easier. They are the GenericCassetteConfigurator and CassetteConfiguratorAdaptor. Javadoc for the ICassetteConfigurator class is provided in `cassette_kit_unzip_dir:/docs/javadoc/configuration`.

GenericCassetteConfigurator class: The custom class may extend the GenericCassetteConfigurator class. The GenericCassetteConfigurator has already implemented all ICassetteConfigurator methods. The custom class may override only the methods it needs to customize. If the add() or remove() methods are overridden, the custom class will be responsible for creating or deleting the cassette's database tables.

CassetteConfigurator Adaptor: The custom class may extend the abstract class CassetteConfiguratorAdaptor. The CassetteConfiguratorAdaptor has implemented some of the ICassette Configurator methods. The custom class will be required to implement all of the following methods:

```
public void add (IDatabase db, Properties properties) throws CassetteCannotBeAddedException;
public void remove (IDatabase db) throws CassetteCannotBeRemovedException;
public void migrate (IDatabase db) throws CassetteCannotBeMigratedException;
```

IDatabase class

The IDatabase class is a "helper" class used by the ConfigClass to access the Payments instance database. The IDatabase interface is passed to the cassette in the add, remove, and migrate methods. The IDatabase interface represents a Payments instance database and does the following:

- It gets the type of the Payments instance database (DB2 or Oracle).
- It gets the user ID of the owner of the Payments instance database tables.
- It gets the user ID of the database user currently connected to the instance database.
- It gets the name of the Payments instance database.
- It connects to the Payments instance database.

To compile your code with this class, use the IDatabase.class file located in the `WC_installdir/payments/lib/eTillConfig.jar` file. Javadoc for the class file is also provided with the Cassette Kit.

Step 3: Deploy your files to the target system

After you have created the final directory structure for your cassette (complete with any customizations if needed), do the following:

1. Compress the cassette directory structure into a ZIP or JAR file.
2. Uncompress the file to the following directory on your target system:

```
WC_installdir/payments/cassettes/cassetteName
```

where `cassetteName` is the name of your cassette.

3. Use the WebSphere Commerce Configuration Manager to add your cassette to the Payments instance.

To see an example of what a cassette file structure looks like, refer to the cassette file structure for one of the IBM-supplied cassettes in the `WC_installdir/payments/cassettes` directory.

Typical installation examples

The following examples illustrate the steps to take to install and add your cassette to WebSphere Commerce Payments.

Scenario for a plain vanilla installation

In this example, the cassette you've written does not require any special Java system properties, required parameters, or any other advanced configuration. To install and add your cassette, ensure that your cassette installation does the following:

1. It creates a *cassetteName/* directory under the *Payments_installdir/cassettes/* directory. The cassette name directory reflect the name of your cassette.
2. It creates the following directories under the *cassetteName* directory:
 - lib/
 - pspl/
 - schema/
3. It provides the *cassette_properties.xml* file in the *cassetteName* directory.
4. It provides a JAR file with cassette classes in the *lib/* directory.
5. It provides a PSPL file in the *pspl/* directory.
6. It provides create and drop database table scripts in the *schema/* directory.

This is all your cassette has to do to get installed. With this information in the appropriate directories, the WebSphere Commerce Configuration Manager can add your cassette to any instance.

Scenario for a more complex installation

In this scenario, your cassette requires both system properties and required parameters, and must also modify the database tables after they are created for the instance. Ensure that your cassette installation does the following:

1. It creates a *cassetteName/* directory under the *Payments_installdir/cassettes/* directory. The cassette name directory reflect the name of your cassette.
2. It creates the following directories under the *cassetteName* directory:
 - lib/
 - pspl/
 - schema/
3. It provides the *cassette_properties.xml* file in the *cassetteName* directory. In this example, the XML file includes elements for ConfigClass and RequiredParameters objects.
4. It provides a JAR file with cassette classes in the *lib/* directory.
5. It provides a PSPL file in the *pspl/* directory.
6. It provides create and drop database table scripts in the *schema/* directory.
7. It contains a class that extends the GenericCassetteConfigurator class. The following example illustrates this extension:

```
public class ExampleCassetteConfigurator extends GenericCassetteConfigurator
{
    //This is an example of how you can implement the add method
    //This cassette needs to encrypt the database with a key provided by the user
    public void add( IDatabase db, Map requiredParms){
        //This method is written by you
        createDatabaseTables( db.getType(), db.getConnection);

        String encryptionKey = requiredParms.get("CassetteEncryptionKey");

        //This method is written by you
        encryptDatabase( encryptionKey);
    }
}
```

```
//This is an example of how you can implement the getSystemProperties() method
public Map getSystemProperties(){
    Properties props = new Properties();
    props.add("cassette.ExampleCassette.UseSSL", "1");
    props.add("cassette.ExampleCassette.CassetteTestMode, "0");
    return(props);
}
...
...
...
}
```

This is all your cassette needs to do to install with a custom extension class. With this information in the appropriate directories, the WebSphere Commerce Configuration Manager can add your cassette to any instance.

Migration considerations and steps

In former versions, WebSphere Commerce Payments provided its own migration mechanism. Starting with version 5.5, migration of cassettes is handled through the WebSphere Commerce Instance Migrator (WCIM). The WCIM migrates an old Payments instance to a new Payments instance. Refer to the *WebSphere Commerce Migration Guide* for a description of the WCIM and for instructions on how to migrate to version 5.5 of WebSphere Commerce Payments.

To migrate a third-party cassette or a cassette that you have written, you must implement the migrate() method. The migrate method can be used before or after a WebSphere Commerce Payments migration:

- Before a Payments migration:

After installing WebSphere Commerce Payments Version 5.5, your cassette can be installed in the cassettes directory (*WC_installdir/payments/cassettes/cassetteName*). Then, when a migration of WebSphere Commerce Payments is performed, your cassette will be migrated.

In addition to having your cassette in the proper directory structure before the migration, you must ensure that you do the following:

- You should ensure that your *cassette_properties.xml* file contains a *Cassette:MigrateScript* element, complete with the name of the migrate script that will be used to migrate the cassette tables. If your cassette has database table changes, you should include a *Cassette:MigrateScript* element.
- If you provide a *ConfigClass* class, it should implement the migrate method for anything not covered in the migrate script.
- You must implement the migrate interface in the *ICassetteConfigurator.java* program to migrate your cassette. See “Migrate interface in *ICassetteConfigurator.java*” on page 184 for more information.

- After a Payments migration:

If a Payments migration has already occurred, you can still install and migrate your cassette. The cassette installer should run the *paymentcassettemigrator* script after the Payments migration occurred. For information on how to run the *paymentcassettemigrator* script, see “Running the *paymentcassettemigrator* script” on page 183. This script migrates the cassette by calling the migrate method you provide as the cassette writer. The migrate method is described in “Using the migrate method” on page 183.

Before running the *paymentcassettemigrator* script, you must ensure that you do the following:

- You should ensure that your `cassette_properties.xml` file contains a `Cassette:MigrateScript` element, complete with the name of the migrate script that will be used to migrate the cassette tables. If your cassette has database table changes, you should include a `Cassette:MigrateScript` element.
- If you provide a `ConfigClass` class, it should implement the migrate method for anything not covered in the migrate script.
- You must implement the migrate interface in the `ICassetteConfigurator.java` program to migrate your cassette. See “Migrate interface in `ICassetteConfigurator.java`” on page 184 for more information.

General information about migration events can be viewed in the Instance Migrator log file (`wcim_dir/wcim.log`). In addition, Payments-level logging of migration activity is recorded in the `wcim_dir/Payments.log` file.

Using the migrate method

The migrate method is an API that is called by the migration process. It is used to migrate a cassette that you have written for use with Payment Manager Version 2.2.x, 3.1.2, or WebSphere Commerce Payments Version 3.1.3. It is also used to migrate new versions of IBM-provided payment cassettes.

If you create a `ConfigClass` class (rather than use the `GenericCassetteConfigurator` class), you must be familiar with the syntax for using the migrate method. The syntax for using the migrate method is as follows:

```
migrate(IDatabase database, String curentVersion) throws CassetteDoesNotExistException,
InvalidMigrationException, MigrationErrorException, DatabaseOperationFailedException;
```

If the migrate method is not found during migration, the `GenericCassetteConfigurator` class is used instead. The `GenericCassetteConfigurator` class configures the cassette by running the database migration script for the database type used by the instance.

Running the paymentcassettemigrator script

If an installation and migration of WebSphere Commerce Payments has already occurred, the `paymentcassettemigrator` script can be used to migrate a new version of your cassette. For example, you might need to migrate a cassette you have updated which did not have database table changes. The script migrates the cassette by calling the migrate method previously described. The script is located in the `Payments_installdir/bin` directory and is run from the command line.

Note: Do not attempt to run this script if the WebSphere Commerce Payments installation and migration has not occurred. Your cassette will not be migrated or function properly.

The `paymentcassettemigrator` script does the following:

- It places the cassette’s files in their correct location in the WebSphere Commerce Payments environment.
- It executes the `ConfigClass` class if provided as well as the migrate script if provided. The migrate script is used only if you use or extend the `GenericCassetteConfigurator` class.

Before running this script, ensure that your cassette files are installed into the following directory as described in “Step 1: Create the directory structure” on page 173:

```
wc_installdir/payments/cassettes/cassetteName
```

To run the `paymentcassettemigrator` script, do the following:

1. From an command prompt for your platform, enter the following from the `WC_installdir/payments/cassettes/cassetteName` directory:

```
paymentcassettemigrator cassetteName instance password
```

cassetteName

The name of the cassette as shown in the `cassette_properties.xml` file for the cassette. See the `<Cassette:CassetteName>xxxxx</Cassette:CassetteName>` tag for the value.

instance

The name of the Payments instance where the cassette currently resides.

password

The instance password for updating the cassette information.

After running the script, you should receive a message indicating the success or failure of the script. If the script does not run successfully, the cassette is considered unloadable and the WebSphere Commerce Payments framework will ignore the cassette. The framework is still available to work with other cassettes, however.

Migrate interface in `ICassetteConfigurator.java`

As a cassette writer, you will need to implement the migrate interface in the `ICassetteConfigurator.java` program to migrate your cassette. You can extend the following classes as your needs dictate:

- `ICassetteConfigurator.class`
- `CassetteConfiguratorAdapter.class`
- `GenericCassetteConfigurator.class`

Database considerations and steps

The WebSphere Commerce Configuration Manager creates database tables for your cassette when these conditions are met:

- A `createTables` script is provided in the proper location.
- The `addCassette()` method is not extended by custom code.

If you implement the `add()` method to configure the database, the `createTables` script is not run for the cassette. Instead, your cassette must create its own tables.

To accomplish this, you must access the WebSphere Commerce Payments database directly and issue the necessary SQL calls to create your tables as well as any views required to support the query commands. At this point, you may also populate your administrative tables with any appropriate default values so they will appear during the cassette configuration step later on.

400 For iSeries, your cassette's database tables should be created in the database collection named in the `PaymentsInstance.properties` file. In addition, user `QPYMSVR` must be given `*ALL` authority to the tables you create and `*PUBLIC` authority should be set to `*EXCLUDE` for security.

Each cassette should provide its own database setup script similar to that provided with the sample cassettes. These scripts can be executed using the java TableMgr class from within your installation program as follows:

```
java TableMgr <yourScriptName> <jdbcURL> <jdbcDriver> <dbOwner> <dbPassword>
```

For example, see the files in the *Payments_installdir/archive* directory (/QIBM/ProdData/PymSvr/Java directory for iSeries). Note that different databases support slightly different SQL syntax. The WebSphere Commerce Payments installation programs use syntax in these scripts that is supported by DB2 UDB and then converts it to the appropriate syntax when using other databases. Your installation program will need to do this as well. Alternatively, you can provide a separate script per database product.

If your cassette has database table changes and you need to migrate your cassette, be sure to include a `Cassette:MigrateScript` element in your `cassette_properties.xml` file.

Finally, to tell WebSphere Commerce Payments that your cassette exists and should be loaded, your installation procedure must add your cassette to the framework's cassette configuration table, `ETCASSETTECFG`. The format of this table is as follows:

Table 13. Cassette configuration table (ETCASSETTECFG)

Field name	Syntax	Description
MessagesKey	Character string	This value may be used in the future to support the user interface's display of your cassette-specific information. For now, set this field to NULL. This is an <i>optional</i> field.

Prior to WebSphere Commerce Payments Version 5.5, there were other fields in this table such as `PaymentSystemName`, `CompanyPkgName`, and so on. These fields are now handled automatically.

Uninstall considerations

While the majority of uninstall activities are simply the reversal of install steps, there are a couple of non-obvious steps that are critical to protecting the integrity of the WebSphere Commerce Payments environment:

- Drop all of the cassette-specific database tables if you do not provide a delete script.
- Delete all of the cassette's records from the framework tables using these SQL statements:

```
delete from ETPAYMENT where PaymentType='<<cassetteName>>';
```

```
delete from ETCREDIT where PaymentType='<<cassetteName>>';
```

```
delete from ETORDER where PaymentType = '<<cassetteName>>';
```

```
delete from ETBATCH where PaymentType = '<<cassetteName>>';
```

```
delete from ETCASSETTECFG where PaymentSystemName = '<<cassetteName>>';
```

```
delete from ETPAYSYS CFG where PaymentSystemName = '<<cassetteName>>';
```

```
delete from ETACCOUNTCFG where CassetteName = '<<cassetteName>>';
delete from EEVENTLISTENER where CassetteName = '<<cassetteName>>';
delete from ETBINARYDATA where PaymentType = '<<cassetteName>>';
```

- Your uninstall process must make sure that the cassette structure directory is removed

Note: This deletes not only the cassette's configuration but also all of the financial data that it recorded. Leaving financial data in the database without valid cassette configuration could cause errors in WebSphere Commerce Payments once it is restarted.

Build a working cassette skeleton

The first step is to build enough of the cassette's WebSphere Commerce Payments infrastructure so you can load, start and stop the cassette. For this step you must

- Determine your cassette name and Java package name using the guidelines described in "Name your cassette" on page 157. For the remainder of this chapter, we will use the name "xxx".
- Create your cassette's "mainline" class for WebSphere Commerce Payments. This class will extend `com.ibm.etill.framework.cassette.Cassette` and will be named *your_cassette's_package_name.XxxCassette*. For now, you only need to write:
 - The constructor
 - The initialization and termination methods called in the "System start sequence" on page 41 and "StartCassette internal sequence" on page 43.
 - Enough of the service method to handle `com.ibm.etill.framework.cassette.AdminRequest` objects that contain `START_CASSETTE_TOKEN` and `STOP_CASSETTE_TOKEN`. Any other type of request object or token type should cause the service method to throw an `com.ibm.etill.framework.payapi.ETillAbortOperation` exception with `PRC_COMMAND_NOT_SUPPORTED`, `RC_NONE`. The last step of your `START_CASSETTE_TOKEN` handling code should issue a console message (using one of the methods from the `com.ibm.etill.framework.log.ErrorLog` class) to indicate that your cassette has started successfully. Any other abstract methods should be created as empty methods for now.
- Build a small *xxxCassette.properties* file which will eventually contain your cassette's console messages and return code messages. At this point, this file should only contain the message that indicates that the cassette has started successfully.

Once you've implemented the above, compile the code, make the necessary configuration changes in the WebSphere Commerce Payments database and environment, and then start WebSphere Commerce Payments with your cassette configured in. WebSphere Commerce Payments should start successfully with no Java exceptions and you should see your console message appear in the `activity.log` file.

Complete this step before continuing on to the next.

Build your administration objects

Now that you have a working cassette skeleton, the next thing to do is build your administrative objects as they will exist and be used in WebSphere Commerce Payments. These objects will contain all of your cassette's active configuration information.

Use an iterative approach where you develop and test each of your administration objects one at a time, rather than writing all of the code before attempting to test any of it. In addition, you should develop the external view of each of your administration objects in conjunction with its Payment Servlet counterpart (this will allow you to use the WebSphere Commerce Payments user interface to test your administration code). Therefore, you will find yourself moving back and forth between this step and the subsequent step for each administration object.

In this step, you will:

1. Add code to your *XxxCassette* class to handle the CREATE, MODIFY, START, STOP and DELETE commands associated with the framework administration objects that you will be augmenting. For example, if your cassette will be augmenting `com.ibm.etill.framework.admin.AccountAdmin` objects, then it should be prepared to handle `com.ibm.etill.framework.cassette.AdminRequest` objects that contain `CREATE_ACCOUNT_TOKEN`, `MODIFY_ACCOUNT_TOKEN`, `START_ACCOUNT_TOKEN`, `STOP_ACCOUNT_TOKEN` and `DELETE_ACCOUNT_TOKEN`. Additionally, if your cassette defines its own primary administration objects (`MerchantCassetteObjects` or `SystemCassetteObjects`), then your service method should also handle `AdminRequest` objects with the tokens associated with those objects.
2. Create Java classes that contain the extension information for the framework administration objects that you want to augment. Note that these classes do not subclass (extend) any of the framework classes. Rather, they are standalone classes that your cassette will create, modify and delete as required by the inbound `AdminRequest` objects.
3. If your cassette defines its own primary administration objects (`MerchantCassetteObjects` or `SystemCassetteObjects`), create one Java class for each such object. These classes will extend the `com.ibm.etill.framework.admin.AdminObject` class in order to inherit all of the basic operational attributes of a primary administration object. These objects will be created, updated and deleted as required by the inbound `AdminRequest` objects containing tokens related to `Merchant` and `System CassetteObjects`.

For the most part, the framework will manage the hierarchy of administration objects when starting, stopping or deleting any of its objects. In other words, when a `PaySystemAdmin` object is stopped, the framework ensures that all of its `AccountAdmin` objects are stopped too. For each of the `AccountAdmin` objects, the cassette will be called to stop its own corresponding object. However, if the cassette has defined any `MerchantCassetteObjects` or `SystemCassetteObjects` that "belong to" a given framework administration object, it is the cassette's responsibility to stop and start each of these when it receives the corresponding START or STOP token for the framework object.

Build your external view of administration objects

As stated above, we suggest that you perform this step for each administration object as soon as you've completed the object's implementation for the Payment Servlet.

This is the only time we will discuss the implementation of classes that represent the external view of your cassette's data separately from the implementation of those objects within the Payment Servlet, because you will typically develop both classes at about the same time.

There are actually two aspects to this step:

- Developing the code which will provide the view of your administration data through WebSphere Commerce Payments Query commands
- Developing the Payment Server Presentation Language for your data so that the WebSphere Commerce Payments user interface can display and manipulate the data.

The best approach to writing this code is to copy the corresponding code in the sample cassette and modify it to fit your needs. The Payment Servlet code and PSPL are particularly well-suited to this type of "pattern matching," which you are strongly encouraged to do.

To support the Query commands, do the following:

- Add code to your *XxxCassetteQuery* class to handle the Query commands associated with the framework administrative objects that you will be augmenting. Using the example from the previous step, if your cassette will be augmenting `com.ibm.etill.framework.admin.AccountAdmin` objects, then it should be prepared to handle `com.ibm.etill.framework.xdm.QueryAccountRequest` objects. Note that Query commands only operate on framework objects, there are no `QueryMerchantCassetteObject` or `QuerySystemCassetteObject` commands. This means any of the cassette's `MerchantCassetteObjects` or `SystemCassetteObjects` must be exposed along with the framework administration object with which they are associated. For example, if your cassette defines a set of `MerchantCassetteObjects` named `Brand` within each account, then the external view of those objects should be generated and returned as part of the cassette output for the `QueryAccounts` command. To understand the interactions between the framework and cassette for Query commands, see "Query API sequence" on page 90.
- Create Java classes that represent the external view of your cassette's data which augments each framework administration object:
 - The constructor for these classes should take a `java.sql.ResultSet` object as input.
 - The class should also have a method which translates the cassette object into a `com.ibm.etill.framework.cassette.query.CassetteExtensionObject`. Each data item in the external view will be represented as a `com.ibm.etill.framework.cassette.query.CassetteProperty` object.
 - This class should have a method that will combine the various `CassetteExtensionObjects` and any associated `com.ibm.etill.framework.cassette.query.CassetteConfigObjects` (see below) with the associated framework external view object (for example, the `com.ibm.etill.framework.xdm.PSServerAccount`, or `com.ibm.etill.framework.xdm.PSServerPaymentSystem`).

That these classes do not subclass (extend) any of the framework classes. Rather, they are standalone internal classes that your cassette uses to generate the appropriate `CassetteExtensionObjects` and `CassetteConfigObjects`.

- If your cassette defines its own primary administration objects (`MerchantCassetteObjects` or `SystemCassetteObjects`), then create one Java class

to represent the external view of each such object. These classes will subclass (extend) `com.ibm.etill.framework.xdm.PSServerAdminObject` to pick up the basic framework attributes and methods for generating the external view of primary administration objects. Aside from that, these classes are much like those described in the bullet above, but they generate `CassetteConfigObjects` instead of `CassetteExtensionObjects`.

- Create one class that subclasses (extends) `com.ibm.etill.framework.xdm.QueryRequest` for each type of cassette object for which you will need to execute an SQL QUERY. The superclass here contains methods which your subclass will override in order to provide the correct database TABLE or VIEW name, FROM clause, SELECT statement and WHERE clause for the SQL QUERY command which the superclass's query method will issue.

Also, if you are protecting sensitive data, note that you can use the `getShowSensitiveData()` method to indicate whether the query request should return the cassette's sensitive data to the user. See "Protecting sensitive data" on page 29 for more information.

Remember, you should pattern match all of this code from the sample cassette source code.

Once you've completed this code, your cassette should be prepared to handle Query commands for the associated administrative objects. If you prefer, you can immediately test this code from a small WebSphere Commerce Payments application program that uses the Java Client API Library, as shown in the *WebSphere Commerce Payments Programming Guide and Reference*. However, before you can issue these Query commands or view their results through the WebSphere Commerce Payments user interface, you will have to describe the data to the WebSphere Commerce Payments user interface servlet. This description is provided through the cassettes's PSPL file.

The sample cassette provides a PSPL file that most cassette writers will find contains examples of the most common types of data descriptions. With this information, you should find it fairly easy to build your own PSPL file. Before testing, this file should be copied into the WebSphere Commerce Payments `pspl` directory.

In general, your PSPL file will contain:

- One screen definition for each of the framework objects that your cassette augments. Each `CassetteExtensionObject` that your cassette returns as Query command output will be described by a separate PSPL screen tag with an `extends` parameter. The value of this parameter contains the name of the framework PSPL screen to which the cassette screen is attached.
- One screen definition for each `CassetteConfigObject` that your cassette defines (recall that these objects represent the external view of `MerchantCassetteObjects` and `SystemCassetteObjects`). Each `CassetteConfigObject` that your cassette returns as Query command output will be described by a separate PSPL screen tag with a `configures` parameter. The value of this parameter contains the name of the framework PSPL screen to which the cassette screen is attached.
- Each data item (that is, `CassetteProperty`) within each `CassetteExtensionObject` and `CassetteConfigObject` will be defined as a separate PSPL field tag within the corresponding screen element. Each of these tags contains parameters that identify the property name, its display attributes, and if the property can be modified through the user interface, the protocol data keyword to use when issuing the MODIFY command.

- Messages to display for the different error situations that may be encountered.
- A list of possible action buttons which can be enabled for the screen. This list must be a subset of the actions that the User Interface Servlet supports.

See the sample cassette's PSPL file and the PSPL reference information in this book for more detailed information.

For a typical cassette, if you are pattern matching from the sample cassette source code and if you have a well-defined cassette administration model, you should be able to build and test a working administration infrastructure, complete with full user interface support in a day or two.

Implement your core protocol function

In this step, we assume that your cassette will contain some central class or set of classes that encapsulate (or provide access to) the core logic for interacting with the Financial institution or other back-end processor. Because of the very "customized" nature of this logic, this guide nor the sample cassette code can tell you how to communicate or process the messages that you exchange with the back end. We can, however, offer some advice:

- In this step you will implement many of the design decisions you made to accommodate restart scenarios. For example, the generation, exchange and processing of protocol messages will most likely occur through calls to different methods on these objects.
- These classes should focus primarily on the payment protocol, not the business logic involved in managing WebSphere Commerce Payments objects, which should be left to the cassette's financial object classes. In other words, use object-oriented design principles to ensure the correct encapsulation of the various functions of your cassette.

To test this core function, you can either write scaffolding code within the cassette to drive requests or move on to the next steps to allow the cassette's financial objects to drive actual WebSphere Commerce Payments requests through. In many cases, both approaches can be used for different portions of the code.

Implement the basic `CassetteOrder`

Now you can begin building your cassette's financial objects by creating a basic cassette-specific order object. This is a Java class, typically named `xxxOrder` that implements the `com.ibm.etill.framework.cassette.CassetteOrder` interface. This class will contain your cassette-specific order information. The primary responsibilities of this class are to:

- Process the request objects which operate directly on Orders, which are:
 - `AcceptPayment`
 - `ReceivePayment`
 - `CloseOrder`
 - `CancelOrder`
 - any protocol requests related to `ReceivePayment` (if supported)
- Manage the database and in-memory versions of the cassette-specific order information. The database versions are supported through the `com.ibm.etill.framework.archive.Archivable` interface, which `CassetteOrder` implements.

- Manage the state of its associated framework `com.ibm.etill.framework.payapi.Order` object as well as the committal of that object to the database using the thread's `com.ibm.etill.framework.archive.CommitPoint` object.
- Provide essential information to the framework through the abstract methods defined by the `CassetteOrder` interface.

In this step, you will build enough of the order processing code to be able to successfully handle valid and invalid `AcceptPayment` or `ReceivePayment` commands (at least one of these commands must be supported by every cassette). You will complete the development of this class in a later step.

To support the `AcceptPayment` and `ReceivePayment` requests:

- Add code to your `XxxCassette` class to:
 - Support the `com.ibm.etill.framework.cassette.AcceptPaymentRequest` or `com.ibm.etill.framework.cassette.ReceivePaymentRequest` (or both, depending upon your cassette design) including the validation and storage of any protocol data that your payment protocol requires. This data is typically kept in the `com.ibm.etill.framework.cassette.CassetteOrder` object. For now, your code should ignore the `APPROVEFLAG` and `DEPOSITFLAG` parameters.
 - Support the `newCassetteOrder` and `resurrectCassetteOrder` methods
- Create the `xxxOrder` Java class as described above. You will need to implement the constructor and all abstract methods, including those defined by the `Archivable` interface.
- At this point, you will probably also need to develop the code that chooses an account for each order. For `AcceptPayment`, this should be a fairly straightforward process. For `ReceivePayment`, it may be a bit more involved. Again, this all depends upon your payment protocol.
- If you are handling `ReceivePayment`, you must also choose or build your `com.ibm.etill.framework.io.ComPoint` subclasses and `com.ibm.etill.framework.cassette.ProtocolRequest` subclasses, writing the code to handle the asynchronous flows involved with these `ComPoint` and managing the intermediate cassette-specific states that go along with this process. The protocol request management will most likely occur within your `CassetteOrder` class.

For more information on the internal sequence that occurs for these commands, see “`ReceivePayment` API sequence” on page 69.

In addition to the code described above, it is now time to write the classes that will support the `QueryOrders` command, as well as any PSPL updates you may need to display any cassette-specific data items on the user interface `Order Details` screens. Use the same approach as you did for returning and displaying your administrative objects.

After completing the code for this step, test your code by writing some simple WebSphere Commerce Payments application programs using the Java Client API Library. Your code should successfully handle valid and invalid `AcceptPayment` or `ReceivePayment` calls. Valid calls should cause a new `Order` object to be created along with an instance of your `xxxOrder` object and both of these should be committed to the database in the appropriate state.

For more information on the framework's `Order` object, its state values and the cassette's responsibilities in maintaining its state, see “`Orders`” on page 13.

Implement basic cassette payments

Now that you can create orders through the WebSphere Commerce Payments API, it's time to move on to payments and your first interaction between your core protocol processor and the WebSphere Commerce Payments objects. For this step, we will implement basic cassette payment support. Here you will create a Java class, typically named *xxxPayment* that implements the `com.ibm.etill.framework.cassette.CassetteTransaction` interface. This class will contain your cassette-specific payment information. The primary responsibilities of this class are to:

- Process the request objects that operate on Payments, which are:
 - Approve
 - ApproveReversal
 - Deposit
 - DepositReversal
- Interact with the cassette's core protocol logic to cause the exchange of any protocol messages that may be required to handle the above commands.
- Handle any communication failures or pending conditions in a manner that allows for timely response to the WebSphere Commerce Payments application's original API request and, if appropriate, later retries of the protocol messages through the use of `com.ibm.etill.framework.cassette.CassetteWorkItem` objects and the framework's timer queue.
- Manage the database and in-memory versions of the cassette-specific payment information. The database versions are supported through the `com.ibm.etill.framework.archive.Archivable` interface which `CassetteTransaction` implements.
- Manage the state of its associated framework `com.ibm.etill.framework.payapi.Payment` object and, if appropriate, the `com.ibm.etill.framework.payapi.Order` object as well as the committal of those objects to the database using the thread's `com.ibm.etill.framework.archive.CommitPoint` object. In addition, any necessary intermediate cassette-specific states should be maintained within the *xxxPayment* object. Remember that while any of these internal states are in effect, the framework `Payment` and `Order` must still be maintained with one of their own state values (for example, `PAYMENT_PENDING`, and so on).
- Provide any necessary services to the framework through the abstract methods defined by the `CassetteTransaction` interface.

In this step, you will build enough of the payment processing code to be able to successfully handle valid and invalid Approve commands. You will also add support for ApproveReversal if your cassette design calls for it. You will complete the remainder of this class in a later step.

Some of the things you will need to do to support the Approve request:

- Add code to your *XxxCassette* class to:
 - Support the `com.ibm.etill.framework.cassette.ApproveRequest` including the validation of framework parameters as they apply to your payment protocol. Once validated, this request is typically forwarded to the *xxxPayment* object for processing.
 - Support the `newCassettePayment` and `resurrectCassettePayment` methods

- Create the *xxxPayment* Java class as described above. You will need to implement the constructor and all abstract methods, including those defined by the *Archivable* interface.

For more information on the framework's *Payment* object, its state values and the cassette's responsibilities in maintaining its state, see "Payments" on page 16.

In addition to the code described above, you should now to write the classes that will support the *QueryPayments* command in the *Payment Servlet* as well as any PSPL updates you may need to display any cassette-specific data items on the user interface *Payment Details* screens. Use the same approach as you did for returning and displaying your cassette's order data.

After completing the code for this step, test your code either by writing some simple *WebSphere Commerce Payments* application programs using the *Java Client API Library* or by using the *Approve* screens of the *WebSphere Commerce Payments* user interface. Your code should successfully handle valid and invalid *Approve* calls. Valid calls should cause a new *Payment* object to be created along with an instance of your *xxxPayment* object and both of these should be committed to the database in the appropriate state. In addition, all necessary protocol messages should be exchanged with the back end per the payment protocol. In all likelihood, you will have to write some scaffolding code to simulate error conditions, including communication failures and pending conditions at the back end.

Once you have the above functions working as described, you can add support for the *ApproveReversal* command, as well as the *APPROVEFLAG* parameter of the *AcceptPayment* and *ReceivePayment* commands (if your cassette design supports them).

Implement *CassetteBatch*

At this point, you have built most of the infrastructure that you will need to build. It is now time to complete the financial object model and command set. You won't be able to test the code you write in this step yet, but you will need to write this code before you can move on to the remaining steps.

In this step, we will build enough of the cassette's batch support to handle the most commonly-accessed batch functions. Here you will create a Java class, typically named *xxxBatch* that implements the `com.ibm.etill.framework.cassette.CassetteBatch` interface. This class will contain your cassette-specific batch information. The primary responsibilities of this class are to:

- Process the request objects which operate on Batches, which are:
 - *BatchClose*
 - *DeleteBatch*
 - *BatchPurge*
 - *BatchOpen* (we recommend that you do **not** support this request)

In addition, this class must handle:

- *Implicit batch creation*
- Any batch-oriented functions related to processing *Payments* and *Credits*.

- Interact with the cassette's core protocol logic in order to cause the exchange of whatever protocol messages may be required for handling the above commands and functions.
- Handle any communication failures or pending conditions in a manner that allows for timely response to the WebSphere Commerce Payments application's original API request and, if appropriate, later retries of the protocol messages through the use of `com.ibm.etill.framework.cassette.CassetteWorkItem` objects and the framework's timer queue.
- Manage the database and in-memory versions of the cassette-specific batch information. The database versions are supported through the `com.ibm.etill.framework.archive.Archivable` interface which `CassetteBatch` implements.
- Manage the state of its associated framework `com.ibm.etill.framework.payapi.Batch` object and, if appropriate, the related `com.ibm.etill.framework.payapi.Payment` and `com.ibm.etill.framework.payapi.Credit` objects including the committal of those objects to the database using the thread's `com.ibm.etill.framework.archive.CommitPoint` object. In addition, any necessary intermediate cassette-specific states should be maintained within the `xxxBatch` object. Remember that while any of these internal states are in effect, the framework `Payment` and `Order` must still be maintained with one of their own state values (for example, `BATCH_CLOSING`, and so on).
- Provide any necessary services to the framework through the abstract methods defined by the `CassetteBatch` interface.

Write the code necessary to successfully handle the most commonly accessed batch functions. You will complete this class later. Here are some of the things you will need to do to support these functions:

- Add code to your `xxxCassette` class to:
 - Support the `BatchClose` request as described above, including the validation of framework parameters as they apply to your payment protocol. Once validated, this request is typically forwarded to the `xxxBatch` object for processing.
 - Support the `newCassetteBatch` and `resurrectCassetteBatch` methods.
- Create the `xxxBatch` Java class as described above. Implement the constructor and all abstract methods, including those defined by the `Archivable` interface.
- Write a method that performs implicit batch creation as needed according to the requirements of your payment protocol.
- Write any batch-oriented functions your payment protocol requires when executing `Payment`- or `Credit`-oriented commands.

For more information on the internal sequence that occurs for these commands, see "BatchClose API sequence" on page 79.

You should also write the classes which will support the `QueryBatches` command in the `PaymentServlet` as well as any PSPL updates you may need to display any cassette-specific data items on the user interface `Batch Details` screens. Use the same approach as you did for returning and displaying your cassette's order data.

After you have written and compiled this code, move on to the next step.

Complete cassette payment

With your batch class in place, you can add the remaining code to your *xxxPayment* class to support:

- `Deposit`
- `DepositReversal`

These commands typically affect the contents of a batch through the `com.ibm.etill.framework.payapi.Batch` object's `addPayment` and `removePayment` methods, respectively.

Some of the things you will need to do to support the remaining payment-oriented requests:

- Add code to your *XxxCassette* class to support the `com.ibm.etill.framework.cassette.DepositRequest` and `com.ibm.etill.framework.cassette.DepositReversalRequest` requests, including the validation of framework parameters as they apply to your payment protocol. Like previous payment-oriented requests, these requests are typically forwarded to the *xxxPayment* object for processing once they have been validated.
- Update the *xxxPayment* Java class to handle these requests.

For more information on the internal sequence that occurs for these commands, see “Deposit API sequence” on page 74 and “DepositReversal API sequence” on page 75.

After completing the code for this step, test your code either by writing some simple WebSphere Commerce Payments application programs using the Java Client API Library or by using the Payment or Deposit screens of the WebSphere Commerce Payments user interface. Your code should successfully handle valid and invalid `Deposit` and `DepositReversal` calls. Valid calls should cause the appropriate state changes in the framework's Payment object as well as in the associated Batch. All updated objects should be committed to the database in the appropriate states. In addition, all necessary protocol messages should be exchanged with the back end per the payment protocol. In all likelihood, you will have to write some scaffolding code to simulate error conditions, including communication failures and pending conditions at the back end.

For more information on the framework's Payment object, its state values and the cassette's responsibilities in maintaining its state, see “Payments” on page 16.

Implement cassette credits

At this point, you have a fully-functioning cassette from the Payment perspective. Now it's time to add Credit support. Credits typically reflect Payments very closely, with this difference: Credit objects are created during a Refund command (the analog of `Deposit` for Payments) and have no analog to the `Approve` command.

Because of the close similarities between these two types of objects, use your now-complete *xxxPayment.java* class as the model for your *xxxCredit.java* class.

For more information on the internal sequence that occurs for these commands, see “Refund API sequence” on page 76 and “RefundReversal API sequence” on page 77. For more information on the framework's Credit object, its state values and the cassette's responsibilities in maintaining its state, see “Credits” on page 18.

Complete the remaining transactional support

Go back and write the code to support the remaining API commands that your cassette design supports, but that you have not yet implemented. For all commands that your cassette design does **not** support, the cassette should throw a `com.ibm.etill.framework.payapi.ETillAbortOperation` exception with `PRC_COMMAND_NOT_SUPPORTED`, `RC_NONE`. (Recall the recommendation made earlier. If a command is not directly supported by your payment protocol but can be simulated, then accept it. If the unsupported command cannot be simulated, then reject it with `PRC_COMMAND_NOT_SUPPORTED`, `RC_NONE` instead of ignoring its existence. You should expect and tolerate this return code pair as a likely possibility and handle it accordingly.)

It is now time for more extensive reliability and data integrity testing. Some areas to cover are:

- More heavy testing of your logic to restart requests that were pending when the account, payment system, cassette or WebSphere Commerce Payments was stopped.
- Stress testing with multiple concurrent API commands from multiple application threads.
- Reliability of work items that are queued on to the framework's service or timer queues.
- Stress testing batch closure logic (by sending a lot of Payment- and Credit-oriented commands to WebSphere Commerce Payments and frequently close batches).

You should also ensure that you provide a means to create orders with your cassette so that new users of your cassette can easily verify that it was installed correctly. In WebSphere Commerce Payments this can be accomplished by creating a Cashier profile and using the `SampleCheckout` application that is provided with the WebSphere Commerce Payments framework.

Create Cashier profiles (optional)

Cashier profiles make your cassette more easily usable by merchant server software (catalog systems, etc.). A Cashier profile is an XML document that describes how orders should be created for a given cassette. This allows the merchant software writer to concentrate on integrating with WebSphere Commerce Payments in a generic way rather than having to write code that deals with cassette-specific information. Creating a Cashier profile is optional; you can still create WebSphere Commerce Payments orders without using the Cashier via the `AcceptPayment` and `ReceivePayment` API commands. The use of the Cashier, however, is preferred since it allows the potential for the merchant to introduce new cassettes to the system without the need for rewriting any code. To learn about the Cashier component of WebSphere Commerce Payments, please refer to the *WebSphere Commerce Payments Programming Guide and Reference*. This section assumes that you have read the Cashier chapter of the *Programming Guide* and are familiar with the basics of the Cashier.

As a cassette writer, you should provide profiles which illustrate the way(s) in which your cassette can be used. This means that you must come up with meaningful combinations of framework and cassette parameters. If your cassette has multiple groups of protocol-data parameters which are independently meaningful together, you should write a profile for each grouping. For example, if your cassette can use two mutually exclusive fraud-detection mechanisms, you

would write one profile for each mechanism. Similarly, if your cassette supports both wallet-based and merchant-originated purchases, you should write one profile for each of these modes of operation. You should name your profile file in accordance with the guidelines in the *WebSphere Commerce Payments Programming Guide and Reference*.

For example:

```
MerchantSoftwareNameMyCassette.profile , or
```

```
MerchantSoftwareNameMyCassetteFraud1.profile , or
```

```
MerchantSoftwareNameMyCassetteFraud2.profile.
```

To use the Sample Checkout application described in the *WebSphere Commerce Payments Programming Guide and Reference*, your Cashier profiles should be installed into the following directory:

```
Payments_installdir/cassettes/cassetteName/profiles.
```

You will want to write profiles for any merchant server which has a published profile interface. For example, if IBM WebSphere Commerce products publish their profile interface, you will want to take that and tailor it to your cassette. This will involve taking a meaningful set of framework and cassette-specific parameters and filling in the values for these parameters based on the published profile interface.

You will also want to provide profiles which are "generic" (merchant server-neutral). These profiles will be templates which describe the sets of parameters to use with your cassette. These generic Cashier profiles will aid authors of "home-grown" merchant software to write profiles for your cassette. Additionally, you should provide a profile for the WebSphere Commerce Payments SampleCheckout application. The SampleCheckout application's interface is defined in the *WebSphere Commerce Payments Programming Guide and Reference*.

It will be extremely helpful if you provide comments in your profiles which indicate what values are expected for each protocol data parameter. For example:

```
<!-- $MyCassetteParm is a four-digit integer value provided by your bank. -->
<Parameter
  name="$MyCassetteParm"><CharacterText>xxxx</CharacterText></Parameter>
```

is preferable to:

```
Parameter
name="$MyCassetteParm"><CharacterText>xxxx</CharacterText></Parameter>
```

If any of your protocol data parameters need information to be supplied by the shopper, you will need to make changes to the BuyPageInformation element of the profile. The specifics of this step are defined by the merchant server for which you are writing the profile. The mechanics of the BuyPageInformation element should be defined as part of the merchant server's profile interface. If you are writing a generic Cashier profile you may skip this step and simply document in the profile what information is needed from the shopper.

Understand platform-specific issues

Using Java minimizes platform-specific issues, but you should remember:

- WebSphere Commerce Payments is provided on Windows 2000, AIX, Solaris, Linux, iSeries, and z/OS. Which of these platforms do you want your cassette to run on?
- Is the install and configure process different for different platforms?
- Do you have any native code?
- Will you be building and packaging this code for other platforms?

The framework provides some platform-specific methods that might vary by platform. However, all of these methods are hidden behind the methods in the `com.ibm.etill.framework.supervisor.Supervisor` class. The most notable of these methods is `Supervisor.loadLibrary`, which is a method for loading system libraries. Cassettes are required to use this method rather than using the Java `System.loadLibrary()` method.

In addition to operating system differences, there are database differences to consider. WebSphere Commerce Payments supports DB2 and Oracle. JDBC can't mask all the differences. You will need to make plans for testing your cassette with each of the databases you intend to support.

Chapter 5. Testing your cassette

This chapter discusses the steps you should take to do a very basic test of your cassette. It includes information on how to configure, process payments and process batches with your cassette. These steps serve two purposes:

- They define a general guide for testing the basic cassette functionality. Because your cassette design may not support all of the WebSphere Commerce Payments commands or objects (or may support them, but with slightly different behaviors), every cassette writer will have to walk through the sequence and tailor the steps to fit their cassette implementation.
- They serve as a template for the tutorial in your cassette documentation. For more information on cassette documentation and the tutorial, see Chapter 4, “Writing your cassette”, on page 171.

Configuring your cassette

Use the information here to configure your cassette. At this point, you should have:

- Installed and configured the WebSphere Commerce Payments component
- Added your cassette to a Payments instance
- Started WebSphere Commerce Payments
- Started WebSphere Application Server
- Started the Web server
- Defined a WebSphere Commerce Payments user with administrative authority
- Created a merchant and Merchant Administrator for that merchant

To authorize a merchant to use a cassette, you must log onto WebSphere Commerce Payments as a Merchant Administrator or Payments Administrator. For information about administrative tasks, see the *WebSphere Commerce Administration Guide*.

After installing your cassette, you must configure the cassette before you can process customer transactions. This tutorial will show you how to configure your cassette. For detailed information on administration, configuration, and payment functions, see the online Help for the WebSphere Commerce Payments user interface.

Using the tutorial software as a model, this chapter demonstrates everything you must do to achieve a fully functioning cassette. This information walks you through fictitious scenarios that simulate real-world functions. And while you need not complete the entire walk-through, it is important that you complete these tasks to become familiar with the common cassette tasks:

1. Create an account.
2. Create any MerchantCassette or System Cassette objects that your cassette has defined, as necessary.

In addition to the required configuration tasks above, we will walk through common payment-processing tasks.

Starting the WebSphere Commerce Payments user interface

Our first task is enabling a merchant to use your cassette. This must be done by a user with Payments Administrator access.

To start the WebSphere Commerce Payments user interface:

1. Point your browser to `http://host_name:port/webapp/PaymentManager/`, where *host_name* is the host name of the machine running the Web Server for Payments, and *port* refers to the port number Payments is running on as shown in the Configuration Manager WebServer information for your Payments instance.
2. On the WebSphere Commerce Payments Logon window, type the Payments Administrator's user ID and password and click **Logon**.

Perform required configuration on the cassette

If you have defined any cassette extensions to the `CassetteAdmin` object, click the **Cassettes** link in the navigation frame, and then click link to your cassette. This will bring you to the cassette configuration screen, where you will see the fields representing your extensions. Fill these in as appropriate and then click the "Update" button.

Creating a WebSphere Commerce Payments Merchant and authorizing a cassette

If you haven't already created a merchant, you must do that first and authorize that merchant to use a payment cassette. To create a merchant, you must log into the WebSphere Commerce Payments as an administrator:

1. From the navigation frame, click **Merchant Settings**.
2. From the Merchant Settings page, click **Add a Merchant** or create a new merchant with merchant number 123456789.
3. At the next window, you will be prompted to authorize use of your cassette:

Field name	Description
Merchant name	Enter <i>Test Store</i> . This is the name that you assign to the merchant. Its only function is to provide display information in the user interface.
Merchant number	Enter <i>123456789</i> . This is a number that you assign which uniquely identifies the merchant in all transaction data.
Authorized cassettes	Check the box next to <code>xxxCassette</code> . Checking this box authorizes the merchant to use the <code>xxxCassette</code> .

4. Click **CreateMerchant** to save the merchant configuration.
5. You will also have to give the user ID `Merchant Administrator` authority for this merchant.

Logging in as the Merchant Administrator

To log off and log in again:

- From the navigation frame, click **Logoff admin** to return to the main WebSphere Commerce Payments Logon page.

- Type the user ID (with Merchant Administrator authority) and the password and click **OK**.

You are now logged in to the WebSphere Commerce Payments user interface with Merchant administrator authority for the Test Store merchant. For the remainder of the tutorial, you will act as the Merchant administrator. Notice that your view of the WebSphere Commerce Payments user interface is now limited to merchant administration functions, whereas as the WebSphere Commerce Payments administrator, you had a global view of both merchant and WebSphere Commerce Payments administration.

Creating an account

So far, you have enabled one merchant, the Test Store, to use your cassette. Now, you need to establish an *account* for your cassette.

An account is a relationship between the merchant and the financial institution which processes transactions for that merchant. There can be multiple accounts for each payment cassette. But for the purposes of this tutorial, you will create one account for your cassette.

To create an account:

1. From the navigation frame, click **Merchant Cassette Settings**.
2. From the Merchant Cassette Settings page, click the **xxxCassette** icon in the Test Store.
3. From xxxCassette window, click **Accounts**.
4. Click **Add an Account** on the Accounts window.
5. At the next window, you will be prompted to enter the following information (note that the italicized text must be entered in these fields for the tutorial):

Field name	Description
Account name	Enter <i>xxxCassette Account</i> . This is the name that you assign to the account. Its only function is to provide display information in the user interface.
Account number	Enter <i>1</i> . This is a number that you (that is, the hosting service provider or the merchant administrator) assign to uniquely identify this account in all transaction data. Note that if account number <i>1</i> , is already defined for the Test Store merchant, then you may choose another number. We will assume for the remainder of this tutorial, however, that you are using account number <i>1</i> .
any cassette-specific fields	Fill in your cassette-specific fields, as required.

6. Click **Create account** to create the new account.

Creating cassette-specific objects

Now that the account has been created, you may need to create any required MerchantCassetteObjects that will "belong" to the account. Each type of MerchantCassetteObject will be listed as a separate choice following "Account Settings" under the "Settings" column of each individual account's summary screen. Your cassette is responsible for providing the PSPL to manage the user interface for these cassette-defined objects.

If you have defined MerchantCassetteObjects or SystemCassetteObjects under the "Merchant Cassette Settings" screen (that is, the screen that contains the "Accounts" link under the "Settings" column), you will see a similar list of choices for each such object. In this case, create and configure each such object as necessary.

Managing payment processing

As the Merchant Administrator, you have global merchant authority, which means that you can perform:

1. Merchant-specific administration functions
2. All payment processing functions

In a real business scenario, you may choose to delegate payment processing tasks to other merchant-defined users who possess limited payment processing authorities (such as, Supervisor and Clerk). In this tutorial, you, as the Merchant Administrator, will perform these tasks.

Having completed all of the WebSphere Commerce Payments and merchant administration tasks necessary to begin payment processing, you are now ready to start:

- Approving orders
- Depositing payments
- Settling batches
- Issuing credits
- Viewing daily batch totals

For the purposes of this tutorial, you will need to provide a mechanism to create sample orders as needed. WebSphere Commerce Payments provides a SampleCheckout application, which is a simple order entry system that supports multiple payment methods. Each payment method is supported by a Cashier profile, which is an XML document that describes how orders should be created for that payment method. See "Create Cashier profiles (optional)" on page 196 for details on how to create a Cashier profile for your cassette. In addition, see the *WebSphere Commerce Payments Programming Guide and Reference* for details on the SampleCheckout application.

Creating orders using the SampleCheckout application

A real business environment features a customer who creates orders using a merchant's Internet storefront and a merchant who processes payments for those orders using WebSphere Commerce Payments. In order for you to walk through the WebSphere Commerce Payments payment processing functions, you need to create orders that require payment processing. To simulate a merchant's Internet storefront, and facilitate order creation, use the SampleCheckout application in conjunction with your cassette's Cashier profile. To access the SampleCheckout application and create orders:

1. Point your browser to `http://host_name:port/webapp/SampleCheckout/`, where *host_name* is the host name of the machine running the Web Server for Payments, and *port* refers to the port number Payments is running on as shown in the Configuration Manager WebServer information for your Payments instance.
2. At the SampleCheckout window that appears, enter the data that your Cashier profile indicates is pertinent for the creation of orders.
3. Click **Buy**.

Repeat these steps two more times so that you have three orders for which to process payments.

Approving orders

Once you have created three orders using the SampleCheckout application, return to the browser window, where the WebSphere Commerce Payments user interface is displayed.

Note: If you used the same browser window to access the Sample Checkout application, you will need to point your browser once again to the WebSphere Commerce Payments URL (that is, `http://host_name:port/webapp/PaymentManager/`) and login with merchant authority.

To approve an order:

1. From the navigation frame, click **Approve**.
2. From the Approve window, select the check box next to the order that you want to approve and click **Approve**.
3. Click **Approve Selected**. The Approve Results page displays the status of your approve request.
4. When your approval is complete, click **Return to the Approve Screen**.

Two orders are still awaiting approval. You could have approved them all simultaneously (for their full amounts), by clicking **Approve All** from the Approve page. Instead, you will work with each order individually to better demonstrate the many facets of the Approve function.

Approving orders from the Order page

In this section, you will approve an order from the Order page (rather than from the Approve page), but you will approve only part of the total order amount.

1. From the Approve page, click the **Order number** for one of the remaining orders awaiting approval. If your cassette exposes any of its extensions to Orders, you should see them listed in the bottom half of the Order detail screen.
2. From the Order page, you can view order details. Click **Approve** to approve and deposit this order.
3. From the Order Approve window, change the sale amount to **3.00** and click **Approve** to approve this order for three dollars.

When approval processing is complete, the Order page refreshes and displays approval status.

Using the Sale function to approve orders

Because you approved only *part* of the last order you worked with, you still have two order entries in the Approve window. In this exercise, you will use the *sale* function to approve the remaining orders.

The sale function allows you to approve an order and move it directly into deposited state, bypassing approved state. The sale function automatically performs an approve and a deposit on your order payment. (Thus, you can think of sale as Approve with AutoDeposit).

Do the following to approve an order with the sale function:

1. From the navigation frame, click **Approve**.

- From the Approve page, click **Sale All**. When processing is complete, the approval status is displayed for each order submitted for sale.
- When your sale is complete, click **Return to the Approve Screen**.

Depositing payments

Deposit allows you to deposit order payments. A single order number can have multiple payments associated with it. You may see the same order number appear multiple times in the same list, each time with different payment information.

To deposit a payment, do the following:

- From the navigation frame, click **Deposit**.
- Select a box for one of the payments listed and click **Deposit Selected**. If your cassette exposes any of its extensions to Payments, you should see them listed in the bottom half of the Payment detail screen. When processing is complete, success or failure status will appear in the Deposit Results page next to the payment submitted for deposit.
- When your sale is complete, click **Return to the Deposit Screen**.

Tip: You can deposit part of a payment. To deposit part of a payment, do the following:

- From the Deposit page, click the **Payment number** for one of the payments awaiting deposit.
- From the Payment page, click **Deposit**.
- In the Order Payment page, change the deposit amount from 5.00 to 3.00 and click **Deposit** to deposit this payment for three dollars.

Settling batches

A batch is a collection of payments and credits that are processed as a unit by a financial institution. A batch is associated with a merchant and an account. The payments that you deposited in the previous exercise will now appear in a batch. You must settle this batch to initiate processing by the financial institution. The financial institution is responsible for the transfer of funds once settlement is complete.

To settle a batch, do the following:

- From the navigation frame, click **Batch Search**.
- In the Batch Search page, you can enter the following information to narrow your search:

Field name	Description
Merchant	The name of the merchant whose batch you are searching for. If there are less than 500 merchants in WebSphere Commerce Payments database, select the merchant name from the drop-down list. If there are more than 500 merchants in the database, type the name of a merchant.
Batch Number	The number that uniquely identifies the batch within the merchant.
State	The state of the batch: <ul style="list-style-type: none"> • Open • Closed

Field name	Description
Balance Status	The balance status of this batch: <ul style="list-style-type: none"> • Balanced: the batch has been successfully balanced (that is, all totals agree). • Out of balance: an unsuccessful attempt has been made to balance this batch (that is, all totals do not agree).
Payment Type	Identifies the payment cassette or protocol used to place the order. Select the payment type implemented by your cassette.
Batch Open Date	Use the after and before fields below to search for batches opened during the specified range in time: <ul style="list-style-type: none"> • After: Specify a date to search for all batches opened on and after this date. • Before: Specify a date to search for all batches opened on and before this date.
Batch Closed Date	Use the before and after fields below to search for batches closed during the specified range in time: <ul style="list-style-type: none"> • After: Specify a date to search for all batches closed on and after this date. • Before: Specify a date to search for all batches closed on and before this date.

3. Click **Search** to initiate a batch search.
4. Click the batch number to view information about the batch.
5. From the Batch window, you can view useful batch information, including the total number and amount of both payments and credits in the batch. If your cassette exposes any of its extensions to Batches, you should see them listed on the Batch screen. Click **Batch Details** to see a detailed listing of all payments and credits in this batch. You will see the four payments you just created and no credits.
6. Click **Settle** to settle the batch. When processing is complete, settle status is displayed in the Settle Results page.

Issuing a credit

Credits are issued against orders and can be given for any amount.

To issue a credit, do the following:

1. To find the order for which you want to issue credit, from the navigation frame, click **Order Search**.
2. At the Order Search page, you can enter the following information (note that for this tutorial, you will not be entering any parameter information in the fields to narrow your search):

Field name	Description
Merchant	The name of the merchant whose order you are searching for. If there are less than 500 merchants in WebSphere Commerce Payments database, select the merchant name from the drop-down list. If there are more than 500 merchants in the database, type the name of a merchant.
Order Number	A number assigned by the merchant that uniquely identifies the order.
State	The state of the order: <ul style="list-style-type: none"> • Ordered • Refundable • Canceled • Closed
Payment Type	Identifies the payment cassette or protocol used to place the order.
Order Date	Use the after and before fields below to search for orders opened during the specified range in time: <ul style="list-style-type: none"> • After: Specify a date to search for all orders opened on and after this date. • Before: Specify a date to search for all orders opened on and before this date.
Order Amount	<ul style="list-style-type: none"> • Currency: The currency used to place this order. Select the currency type from the drop-down list. • Greater than: Specify a value to retrieve all orders with order amounts that are greater than or equal to the value you specify. • Less than: Specify a value to retrieve all orders with order amounts that are less than or equal to the value you specify.

3. Click **Search**.
4. From the Order Search Results page, click an order number for an order in Refundable state, to view the details of that order.
5. From the Order page, click **Credit** to create a credit against this order.
6. At the Create Credit page, the following information displays:

Field name	Description
Currency	The type of currency used to place this order. This is a read-only field.
Order Amount	The total amount of the order expressed in the currency used to place the order. This is a read-only field.
Approved Amount	This field displays the approved amount. This is a read-only field.
Deposited Amount	This field displays the deposited amount. This is a read-only field.

Field name	Description
Credit Amount	This is the total amount of the order.

Enter the credit amount (any amount up to the deposited amount of the order) and click **Credit**.

When credit processing has completed, the Order page refreshes and displays the credit status. The newly created credit appears under **Credits**.

Viewing batch totals

The last step in this tutorial is viewing daily batch totals. The WebSphere Commerce Payments Reports function allows you to view daily totals for batches in a closed state. To generate a daily batch totals report, do the following:

1. From the navigation frame, click **Reports**.
2. From the Reports page, click **Daily Batch Totals**.
3. In the Batch Totals Report page, type the date for which you would like a batch totals report. Leave this field blank to generate a report for the current date.
4. Type or select the **Merchant name**. If you do not type a Merchant name, a list of all of the batches for the specified date will be displayed. If there are more than 500 batches, only the first 500 batches will be displayed.
5. Click **Search** to generate the batch totals report.

The Daily Batch Totals report computes the totals for all batches that were closed on the date specified on the Search page. These totals are computed on a per-currency basis, so there is one line per currency. Note that these totals cover all payments and credits, not just those made through your cassette.

You have just completed a day in the life of a Payments Administrator and a Merchant Administrator. While individual business models may vary, this tutorial outlines the basic path to establishing a working WebSphere Commerce Payments and demonstrates fundamental payment processing implemented through your cassette. For more information on specific fields in the WebSphere Commerce Payments user interface, see the online help.

Appendix. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION AND CASSETTE KIT "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this publication to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department TL3B/Building 503
PO Box 12195
3039 Cornwallis Road
Research Triangle Park, NC 27709-2195

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

- AIX
- DB2
- IBM
- IBM Payment Server
- iSeries
- OS/390
- pSeries
- S/390

- WebSphere
- z/OS
- zSeries

Microsoft, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Intel, Intel Inside (logos), MMX and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Index

A

- AcceptPayment
 - API sequence diagram 71
 - description 61
- access control 91
- account
 - lock 95
- administrative API commands 39
- administrative object model
 - cassette extension objects 8
 - primary objects 8
- administrative objects
 - AccountAdmin 10
 - Cassette 10
 - CassetteAdmin 10
 - description 7
 - MerchantAdmin 10
 - MerchantCassetteObject 10
 - model 8
 - PayServer 9
 - PaySystemAdmin 10
 - primary 8
 - SystemCassetteObject 10
- API commands
 - administrative
 - sequence diagrams 39
 - payment 61
 - sequence diagrams 67
- Approve
 - API sequence diagram 72
 - description 62
- ApproveReversal
 - API sequence diagram 73
 - description 62
- Archivable interface 102

B

- Background and timed operations 99
- Batch
 - lock 96
 - management
 - Explicit style 21
 - implicit style 21
 - object description 20
 - states
 - BATCH_CLOSED 22
 - BATCH_CLOSING 22
 - BATCH_OPEN 22
 - BATCH_OPENING 22
- BatchClose
 - API sequence diagram 79
 - description 65
- BatchOpen
 - API sequence diagram 78
 - description 64
- BatchPurge
 - API sequence diagram 81
 - description 65

C

- caching, Order and batch 91
- CancelOrder
 - API sequence diagram 83
 - description 64
- Cashier 5
- Cashier profiles 5
- cassette
 - lock 95
 - design 157
 - activities 157
 - commit points 165
 - ComPoints and ETillConnection 167
 - Create scenario diagrams 168
 - documenting 168
 - extensions to framework object model 160
 - database tables 161
 - external view 161
 - identifying 160
 - financial state transitions 167
 - mapping WebSphere Commerce Payments API 164
 - naming your cassette 157
 - restart implications 165
 - properties XML file 174
 - writing 171
 - administration objects 187
 - basic cassette payment 192
 - basic CassetteOrder 190
 - cassette credits 195
 - CassetteBatch 193
 - completing cassette payment 195
 - configuration 175
 - core function 190
 - database setup 184
 - directory structure 173
 - external view administration objects 187
 - installation considerations 173
 - platform-specific issues 197
 - skeleton 186
 - uninstall considerations 185
- CassetteConfiguratorAdaptor class 180
- CassetteControl
 - API sequence diagram 60
 - description 39
- classes, configuration related 178
 - CassetteConfiguratorAdapter 180
 - ICassetteConfigurator 180
 - IDatabase 180
- CloseOrder
 - API sequence diagram 82
 - description 64
- command
 - CassetteControl 39
 - Ignore or reject 36
 - parameters, cassette-specific 31
 - parameters, framework 37
 - Payment API
 - AcceptPayment 61
- command (*continued*)
 - Payment API (*continued*)
 - Approve 62
 - ApproveReversal 62
 - BatchClose 65
 - BatchOpen 64
 - BatchPurge 65
 - CancelOrder 64
 - CloseOrder 64
 - DeleteBatch 65
 - Deposit 62
 - ReceivePayment 61
 - Refund 63
 - RefundReversal 63
 - processing 31
 - processing,
 - Query 90
 - processing, overview 25
 - query, overview 26
- command processing
 - payment and administrative, overview 25
- commands
 - administrative API
 - sequence diagrams 39
 - framework 24
 - payment API 61
 - sequence diagrams 67
 - Payment API
 - DepositReversal 63
- commit points, designing 165
- CommitPoint 102
- ComPoints and ETillConnection
 - designing 167
- Configuration Manager
 - adding cassettes 125
 - handling of cassette 179
 - setting minimum access role 29
- configuring cassettes 178
- considerations, restart implications 165
- control, access 91
- create scenario diagrams
 - designing 168
- CreateAccount
 - API sequence diagram 45
- CreateBatch
 - internal sequence diagram 85
- CreateMerchant
 - API sequence diagram 55
- CreateMerchantCassetteObject
 - API sequence diagram 58
- CreatePaySystem
 - API sequence diagram 50
- CreateSystemCassetteObject sequence
 - API sequence diagram 59
- Credit
 - object description 18
 - states
 - CREDIT_CLOSED 20
 - CREDIT_DECLINED 20
 - CREDIT_PENDING 20

Credit (*continued*)
states (*continued*)
 CREDIT_REFUNDED 19
 CREDIT_RESET 19
 CREDIT_VOID 20
custom extension class 179

D

data model, exported 22
Database access 102
database setup 184
debug tracing 119
DeleteAccount
 API sequence diagram 47
 internal sequence diagram 49
DeleteBatch
 API sequence diagram 80
 description 65
DeleteMerchant
 API sequence diagram 57
DeleteMerchantCassetteObject
 API sequence diagram 58
DeletePaySystem
 API sequence diagram 52
 internal sequence diagram 54
DeleteSystemCassetteObject sequence
 API sequence diagram 59
deploying cassettes 180
Deposit
 API sequence diagram 74
 description 62
DepositReversal
 API sequence diagram 75
 description 63
designing your cassette 157
diagnostic trace 120
documenting cassette design 168

E

encoding 133
error handling
 fatal errors 38
 General runtime errors 38
 unsupported commands 38
 while processing protocol
 messages 38
Error logging 115
event notification 105
explicit batch management
 See implicit batch management
exported data model 22

F

financial object
 Batches 20
 Credits 18
 description 7
 Orders 13
 Payments 16
 states 11
financial state transitions
 designing 167

framework
 command parameters, supporting 37
 commands 24
 Javadoc 154
 lock 95
 object model 7
 responsibilities 91

G

GenericCassetteConfigurator class 179

H

help file 133
hierarchy, Request classes 27

I

ICassetteConfigurator class 178, 180
IDatabase class 180
implicit batch management
 See explicit batch management
independent credit 14, 16
installation
 considerations 173
 examples 181
 steps 173
 uninstall considerations 185
interface, Archivable 102
interface, Restorable 102
internal sequence diagram 86, 87, 88, 89
internationalization 133, 158

J

Javadoc
 classes cassettes extend 154
 classes operated on by cassettes 155
 classes providing framework
 services 155
 framework 154
 framework classes, cassette view
 of 154
 interfaces cassettes implement 154
JRas components 120

L

languages 133
lock
 Account 95
 Batch 96
 cassette 95
 framework 95
 MerchantAdmin 95
 Order 96
 read 95
 write 95
logging, Error 115

M

MerchantAdmin lock 95
messages, Receiving protocol 102
migrate method 183
migrating cassettes 182
model
 administrative object 8
 exported data 22
 framework object 7
ModifyAccount
 API sequence diagram 46
ModifyCassette
 API sequence diagram 42
ModifyMerchant
 API sequence diagram 56
ModifyMerchantCassetteObject
 API sequence diagram 58
ModifyPaySystem
 API sequence diagram 51
ModifySystemCassetteObject sequence
 API sequence diagram 59

N

Notices 209
notification, event 105

O

object
 Batch, description 20
 Credit, description 18
 model
 administrative 7, 8
 administrative, cassette
 extensions 8
 administrative, primary objects 8
 financial 7
 framework 7
 Order, description 13
 Payment, description 16
operations, Background and timed 99
Order
 lock 96
 object description 13
 states
 ORDER_CANCELED 15
 ORDER_CLOSED 16
 ORDER_ORDERED 14
 ORDER_PENDING 15
 ORDER_REFUNDABLE 15
 ORDER_REJECTED 15
 ORDER_REQUESTED 14
 ORDER_RESET 14
order and batch caching 91
overview
 command processing 25

P

parameter
 supporting framework command 37
parameters
 validation 96

- Payment
 - API commands 61
 - sequence diagrams 67
 - object description 16
 - protocol mapping 66
 - states
 - PAYMENT_APPROVED 17
 - PAYMENT_CLOSED 18
 - PAYMENT_DECLINED 18
 - PAYMENT_DEPOSITED 17
 - PAYMENT_EXPIRED 18
 - PAYMENT_PENDING 18
 - PAYMENT_RESET 17
 - PAYMENT_VOID 18
 - paymentcassettemigrator script 183
 - profiles, Cashier 5
 - protecting sensitive data 29
 - protocol mapping, payment 66
 - Protocol message
 - API sequence diagram 84
 - protocol messages, Receiving 102
 - publications, related vii
 - purchasing cards 112

Q

- Query
 - API sequence diagram 90
 - command processing 90

R

- Read lock 95
- realms
 - tracing 120
- ReceivePayment
 - API sequence diagram 69
 - description 61
- Receiving protocol messages 102
- Refund
 - API sequence diagram 76
 - description 63
- RefundReversal
 - API sequence diagram 77
 - description 63
- request classes
 - hierarchy 27
 - overview 27
- responsibilities, framework 91
- restart implications 165
- Restorable interface 102
- RetrieveBatch 86
- RetrieveOrder 87

S

- sensitive data, hiding 23
- sequence diagram
 - AcceptPayment 71
 - Approve 72
 - ApproveReversal 73
 - BatchClose 79
 - BatchOpen 78
 - BatchPurge 81
 - CancelOrder 83
 - CassetteControl 60

- sequence diagram (*continued*)
 - CloseOrder 82
 - CreateAccount 45
 - CreateBatch (internal) 85
 - CreateMerchant 55
 - CreateMerchantCassette object 58
 - CreatePaySystem 50
 - CreateSystemCassette object 59
 - DeleteAccount 47
 - DeleteAccount (internal) 49
 - DeleteBatch 80
 - DeleteMerchant 57
 - DeleteMerchantCassette object 58
 - DeletePaySystem 52
 - DeletePaySystem (internal) 54
 - DeleteSystemCassette object 59
 - Deposit 74
 - DepositReversal 75
 - ModifyAccount 46
 - ModifyCassette 42
 - ModifyMerchant 56
 - ModifyMerchantCassette object 58
 - ModifyPaySystem 51
 - ModifySystemCassette object 59
 - Protocol message 84
 - Query 90
 - ReceivePayment 69
 - Refund 76
 - RefundReversal 77
 - RetrieveBatch (internal) 86
 - RetrieveOrder (internal) 87
 - Service queue (internal) 88
 - StartAccount (internal) 48
 - StartCassette (internal) 43
 - StartPaySystem (internal) 53
 - StopAccount (internal) 48
 - StopCassette (internal) 44
 - StopPaySystem (internal) 53
 - System start 41
 - Timer queue (internal) 89
- Service queue 88
- StartAccount
 - internal sequence diagram 48
- StartCassette
 - internal sequence diagram 43
- StartPaySystem
 - internal sequence diagram 53
- states
 - Batch
 - BATCH_CLOSED 22
 - BATCH_CLOSING 22
 - BATCH_OPEN 22
 - BATCH_OPENING 22
 - Credit
 - CREDIT_CLOSED 20
 - CREDIT_DECLINED 20
 - CREDIT_PENDING 20
 - CREDIT_REFUNDED 19
 - CREDIT_RESET 19
 - CREDIT_VOID 20
 - Order
 - ORDER_CANCELED 15
 - ORDER_CLOSED 16
 - ORDER_ORDERED 14
 - ORDER_PENDING 15
 - ORDER_REFUNDABLE 15
 - ORDER_REJECTED 15

- states (*continued*)
 - Order (*continued*)
 - ORDER_REQUESTED 14
 - ORDER_RESET 14
 - Payment
 - PAYMENT_APPROVED 17
 - PAYMENT_CLOSED 18
 - PAYMENT_DECLINED 18
 - PAYMENT_DEPOSITED 17
 - PAYMENT_EXPIRED 18
 - PAYMENT_PENDING 18
 - PAYMENT_RESET 17
 - PAYMENT_VOID 18
 - StopAccount
 - internal sequence diagram 48
 - StopCassette
 - internal sequence diagram 44
 - StopPaySystem
 - internal sequence diagram 53
 - synchronization 94
 - System start
 - sequence diagram 41

T

- threading 92
- timed operations 99
- Timer queue 89
- tracing, debug 119
- trademarks 210

U

- uninstalling cassettes 185

V

- validation, parameter 96

W

- Web sites viii
- wpm.MinSensitiveAccessRole
 - parameter 29
- write lock 95
- writing your cassette 171
 - administration objects 187
 - basic cassette payment 192
 - basic CassetteOrder 190
 - cassette credits 195
 - CassetteBatch 193
 - completing cassette payment 195
 - core function 190
 - database setup 184
 - external view administration
 - objects 187
 - installation considerations 173
 - platform-specific issues 197
 - skeleton 186
 - uninstall considerations 185



Printed in U.S.A.