Customizing and Extending Oracle Siebel ePayment Manager

Version 4.7

May 31, 2007

ORACLE

Copyright © 1996, 2007 Oracle. All rights reserved.

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

PRODUCT MODULES AND OPTIONS. This guide contains descriptions of modules that are optional and for which you may not have purchased a license. Siebel's Sample Database also includes data related to these optional modules. As a result, your software implementation may differ from descriptions in this guide. To find out more about the modules your organization has purchased, see your corporate purchasing agent or your Siebel sales representative.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS. Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software--Restricted Rights (June 1987). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and we disclaim liability for any damages caused by such use of the Programs.

Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

Contents

1 Preface

About Customer Self Service and eaSuite™

Oracle has developed the industry's most comprehensive software and services for deploying Customer Self-Service solutions. **eaSuite**™ combines electronic presentment and payment (EPP), order management, knowledge management, personalization and application integration technologies to create an integrated, natural starting point for all customer service issues. eaSuite's unique architecture leverages and preserves existing infrastructure and data, and offers unparalleled scalability for the most demanding applications. With deployments across the healthcare, financial services, energy, retail, and communications industries, and the public sector, eaSuite powers some of the world's largest and most demanding customer self-service applications. eaSuite is a standards-based, feature rich, and highly scalable platform, that delivers the lowest total cost of ownership of any self-service solution available.

eaSuite consists of four product families:

- Electronic Presentment and Payment (EPP) Applications
- Advanced Interactivity Applications
- Enterprise Productivity Applications
- Development Tools

Electronic Presentment and Payment (EPP) Applications are the foundation of Oracle’s Customer Self-Service solution. They provide the core integration infrastructure between organizations’ backend transactional systems and end users, as well as rich e-billing, e-invoicing and e-statement functionality. Designed to meet the rigorous demands of the most technologically advanced organizations, these applications power Customer Self-Service by managing transactional data and by enabling payments and account distribution.

- **eStatement Manager™** is the core infrastructure of enterprise Customer Self-Service solutions for organizations large and small with special emphasis on meeting the needs of organizations with large numbers of customers, high data volumes and extensive integration with systems and business processes across the enterprise. Organizations use eStatement Manager with its data access layer, composition engine, and security, enrollment and logging framework to power complex Customer Self-Service applications.
- **ePayment Manager™** is the electronic payment solution that decreases payment processing costs, accelerates receivables and improves operational efficiency. ePayment Manager is a complete payment scheduling and warehousing system with real-time and batch connections to payment gateways for Automated Clearing House (ACH) and credit card payments, and payments via various payment processing service providers.

Oracle’s **Development Tools** are visual development environments for designing and configuring Oracle’s Customer Self-Service solutions. The Configuration Tools encompass data and rules management, workflow authoring, systems integration, and a software development kit that makes it easy to create customer and employee-facing self-service applications leveraging eaSuite.

About This Guide

This document provides information about customizing and extending the features provided by ePayment Manager.

The ePayment Manager SDK module consists of:

- This document, installed when you install the SDK
- JavaDoc for the ePayment Manager APIs, installed when you install the SDK
- java code for the ePayment Manager plug-in samples, provided in this document
- The sample application: *paymentComplex.ear*, installed when you installed ePayment Manager

Related Documentation

This guide is part of the ePayment Manager documentation set. For more information about implementing your ePayment Manager application, see one of the following guides:

Print Document	**Description**
Installation Guide for Oracle Siebel ePayment Manager	How to install and configure ePayment Manager on your system.
Administration Guide for Oracle Siebel ePayment Manager	How to configure and operate the production environment. It describes configuration tasks done after installation.
Administration Guide for Oracle Siebel eStatement Manager	How to set up and run a live eStatement Manager application in a J2EE environment.
Deploying and Customizing J2EE Applications Guide for Oracle Siebel eStatement Manager	How to customize J2EE Web applications for deployment with the eaSuite.
SDK Guide for Oracle Siebel eStatement Manager	How to use the Oracle Software Developers Kit to write custom code against Oracle applications.

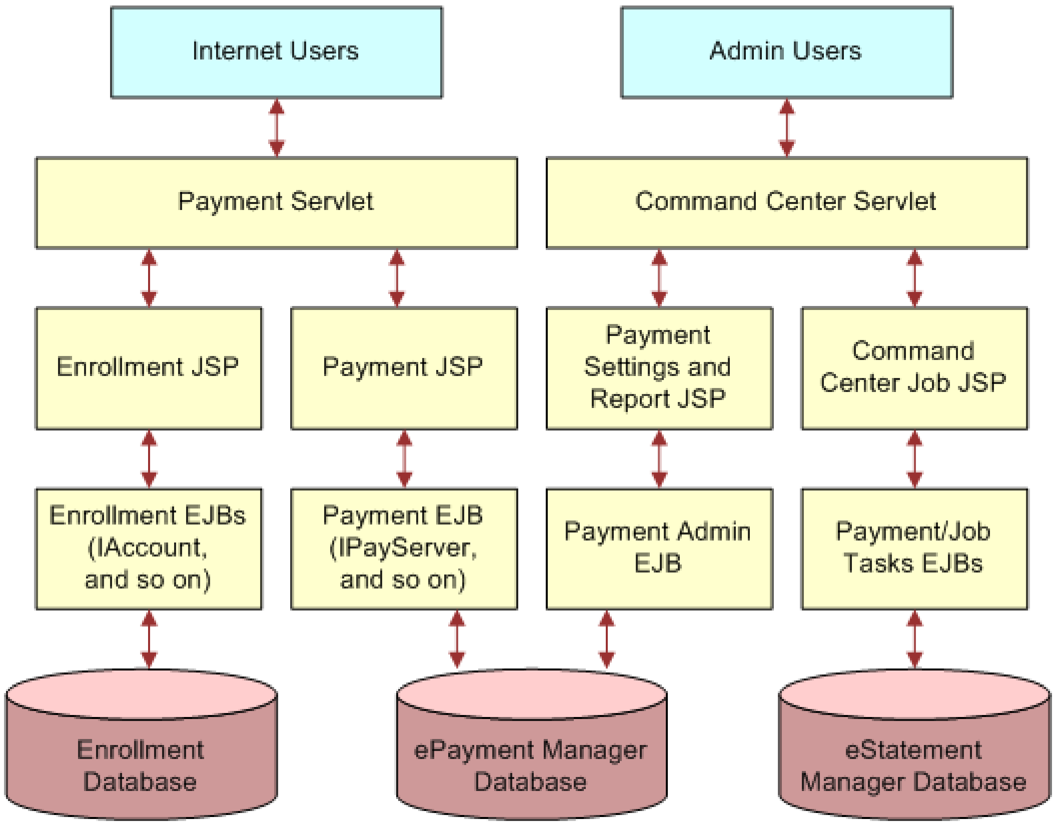
2 Introduction

Architecture of ePayment Manager

ePayment Manager is based on the J2EE platform. It uses Servlets and JSPs for the presentation layer and uses enterprise java beans (EJB) for the business logic layer. It offers the following sets of functions:

- **Enrollment functions**: to enroll users for both viewing bills (eStatement Manager) and paying bills (ePayment Manager). Examples of user information include account numbers and email addresses, and examples of payment account information include bank account numbers and credit card accounts.
- **ePayment Manager functions**: to make payments, set up payment reminders and recurring payments, etc.
- **Administration functions**: to set up payment jobs, view payment reports and configure ePayment Manager Settings.

The following diagram shows an overview of the J2EE architecture of ePayment Manager:



In this architecture, the servlet is responsible for user authentication. After authentication, the servlet forwards the request to JSP pages, which do the bulk of the actual work. The ePayment Manager user JSP pages can be categorized into two groups:

- Enrollment JSP pages are responsible for ePayment Manager user enrollment
- ePayment Manager JSP pages are responsible for core ePayment Manager functionality: schedule payment, set up recurring payment, etc.

All ePayment Manager database access is done through EJB objects. The JSPs and servlets do not access the database directly.

There are also ePayment Manager batch jobs that run inside the eStatement Manager Command Center. For a list and description of ePayment Manager jobs, refer to the *Administration Guide for Oracle ePayment Manager*.

What's in the ePayment Manager Package

The ePayment Manager package contains an ePayment Manager user application called *ear-payment-complex.ear.*

ear-payment-complex.ear allows each user ID to enroll with multiple billers, with multiple user accounts for each biller.

The following table describes the contents of the EAR file:

File	**Description**
META-INF/MANIFEST.MF	The manifest file of this EAR file.
META-INF/application.xml	Lists the J2EE components (EJB JARs and WARs) in this EAR file.
war-payment-complex.war	The WAR files include ePayment Manager servlet and JSP pages for enrollment and payment functionality.
ejb-payment-payserver.jar	ePayment Manager EJB JAR file containing the `IPayServer` EJB bean.
ejb-payment-acctmgr.jar	ePayment Manager EJB JAR file containing the `IPaymentAccountManager` EJB bean.
ejb-payment-query.jar	ePayment Manager EJB JAR file containing the `IPaymentQuery` EJB bean.
ejb-payment-pscustom.jar	ePayment Manager EJB JAR file reserved for Payment internal use.
lib/edx_common.jar	JAR file containing the eStatement Manager's support and utility classes that can be used both by the EJB clients and the EJBs.
lib/payment_client.jar	JAR file containing the ePayment Manager's support and utility classes that an EJB client may find useful.

File	**Description**
lib/payment_common.jar	JAR file containing the eStatement Manager's support and utility classes that can be used both by the EJB clients and the EJBs.
lib/payment_custom.jar	JAR file containing all the ePayment Manager customization classes, such as plug-ins etc. done outside core ePayment Manager.
lib/Verisign.jar	JAR file for Verisign cartridge.

All other JAR files not listed in the table come from eStatement Manager.

In addition to the EAR files listed previously, there is a second EAR file, *ear-payment.ear*, which is provided by the ePayment Manager installation. This EAR file includes the EJB JAR files and WAR file for the ePayment Manager functionality of the Command Center. This EAR cannot be deployed independently, and so it must be merged into *ear-eStatement.ear*. For details about merging these files, see the *Installation Guide for Oracle Siebel ePayment Manager*. Also see the *Deploying and Customizing J2EE Applications Guide for Oracle Siebel eStatement Manager*.

Major ePayment Manager Beans

The following tables describe the major ePayment Manager beans defined in both user EAR and Command Center EAR (*ear-payment.ear*).

Name	PayServer
Remote Interface	*Com.edocs.payment.remote.IPayServer*
Home Interface	*Com.edocs.payment.remote.IPayServerHome*
Bean Type	State-less
Jar file	*ejb-payment-payserver.jar*
Description	This is the main EJB bean for user application to access the ePayment Manager database.

Name	PayAdmin Server
Remote Interface	*Com.edocs.payment.remote.IPayAdminServer*
Home Interface	*Com.edocs.payment.remote.IPayAdminServerHome*
Bean Type	State-less
Jar file	*ejb-payment-admin.jar*
Description	This is the main EJB bean for Command Center to configure ePayment Manager Settings and view payment reports.

Name	IPaymentAccount Manager
Remote Interface	*Com.edocs.payment.remote.PaymentAccountManager*
Home Interface	*com.edocs.payment.remote. IPaymentAccountManagerHome*
Bean Type	Stateful
Jar file	*ejb-payment-acctmgr.jar*
Description	This is the main EJB bean for user application to access payment account information inside ePayment Manager database.

Name	CreditCardSubmit
Remote Interface	*Com.edocs.pwc.tasks.ITask*
Home Interface	*Com.edocs.pwc.tasks.ITaskHome*
Bean Type	Stateful
Jar file	*ejb-payment-ccsubmit.jar*
Description	Credit card submit task.

Name	ChkSubmit
Remote Interface	*com.edocs.pwc.tasks.ITask*
Home Interface	*com.edocs.pwc.tasks.ITaskHome*
Bean Type	Stateful
Jar file	*ejb-payment-chksubmit.jar*
Description	Check submit task.

Name	ChkUpdate
Remote Interface	*com.edocs.pwc.tasks.ITask*
Home Interface	*com.edocs.pwc.tasks.ITaskHome*
Bean Type	Stateful
Jar file	*ejb-payment-chkupdate.jar*
Description	Check update task.

Name	ConfirmEnroll
Remote Interface	*com.edocs.pwc.tasks.ITask*
Home Interface	*com.edocs.pwc.tasks.ITaskHome*
Bean Type	Stateful
Jar file	*ejb-payment-confirm-enroll.jar*
Description	Confirm enroll task.

Name	NotifyEnroll
Remote Interface	*com.edocs.pwc.tasks.ITask*
Home Interface	*com.edocs.pwc.tasks.ITaskHome*
Bean Type	Stateful
Jar file	*ejb-payment-notify-enroll.jar*
Description	Notify enroll task.

Name	RecurPayment
Remote Interface	*com.edocs.pwc.tasks.ITask*
Home Interface	*com.edocs.pwc.tasks.ITaskHome*
Bean Type	Stateful
Jar file	*ejb-payment-recur-pmt_scheduler.jar* and *ejb-payment-recur-pmt_synchronizer.jar*
Description	Recurring payment task.

Name	PaymentReminder
Remote Interface	*com.edocs.pwc.tasks.ITask*
Home Interface	*com.edocs.pwc.tasks.ITaskHome*
Bean Type	Stateful
Jar file	*ejb-payment-reminder.jar*
Description	Payment reminder task.

Name	SubmitEnroll
Remote Interface	*com.edocs.pwc.tasks.ITask*
Home Interface	*com.edocs.pwc.tasks.ITaskHome*
Bean Type	Stateful
Jar file	*ejb-payment-submit-enroll.jar*
Description	Submit enroll task.

3 ePayment Manager Enrollment

Overview

After installing ePayment Manager, you must design and choose the method of enrolling a user. ePayment Manager enrollment is a very important and complex part of ePayment Manager, which this chapter explains in detail.

ePayment Manager enrollment enables a user to enroll with the ePayment Manager system to view and pay bills. Since ePayment Manager includes the ability to view bills, you must use ePayment Manager to login when you have both eStatement Manager and ePayment Manager installed in your system.

ePayment Manager enrollment information consists of two parts:

- **User account information** - includes user name, password, account number, email address, etc., which is used for authentication and bill presentment
- **ePayment Manager account information** - includes check account number and credit card number, which is used to make a payment

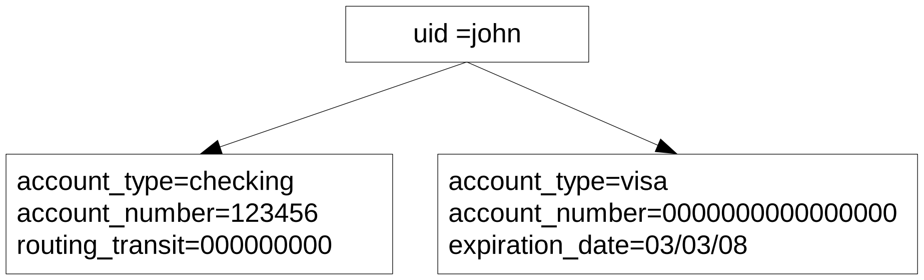
These two groups of information can be in the same database or can be in different databases.

Planning your enrollment model can be complicated, especially the user account part. There are two enrollment models that come with ePayment Manager, which can serve as a starting point in implementing your own enrollment model.

Default ePayment Manager Enrollment Models

The ePayment Manager sample application paymentComplex, save payment account information in the *payment_accounts* table of the ePayment Manager database. Note that users can enroll an arbitrary number of payment accounts (of either check or credit card account).

The following diagram depicts a user "john" with two payment accounts (one check account and one credit card account).

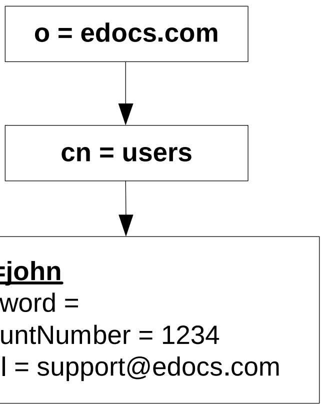


User account enrollment is more complicated and is further explained in the following section.

Single-DDN Model

In this model, a single user can have only one account number per DDN. This model is compatible with the eStatement Manager Sample application. Therefore, a user who enrolls through Sample should be able to login a payment front-end application which uses this model, and vice versa.

There are different ways to implement this enrollment model. One implementation is based on a JNDI-compatible directory like eStatement Manager CDA and has following directory schema:



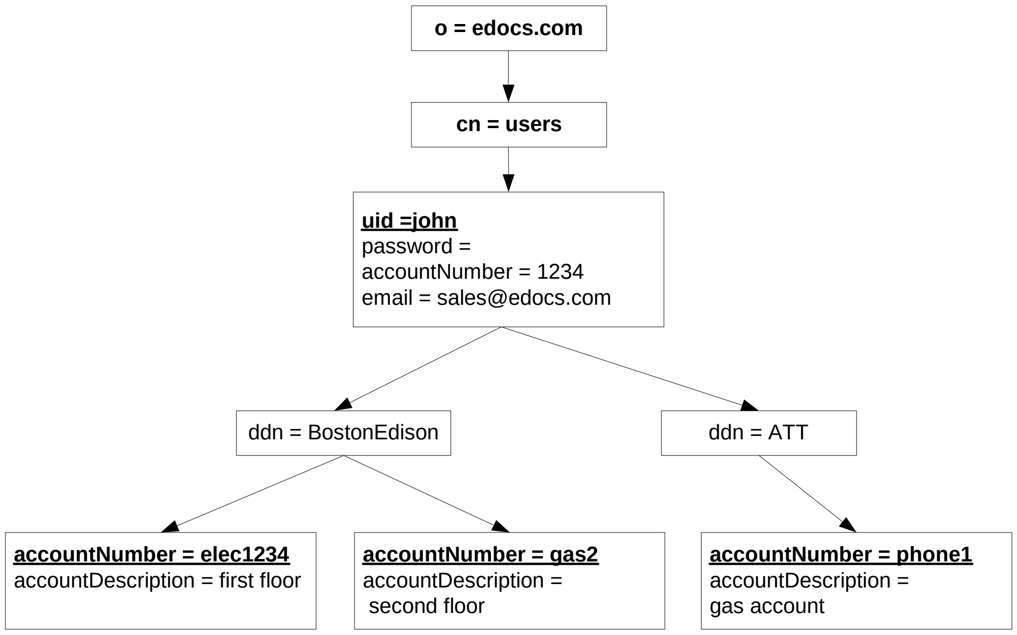
In preceding example, user "john" can have only one account with the biller. The biller name is passed in through URL as "ddn".

Multiple-DDN Model

This model is demonstrated by the paymentComplex sample application, which is packaged as *ear-payment-complex.ear*. It can be accessed using http://host:port/paymentComplex/Payment?app=Payment.

In this model, a single user can enroll with multiple billers, and can have more than one account with each biller. This essentially implements a hierarchy of user accounts.

There are different ways to implement this enrollment model. The paymentComplex implementation is based on a JNDI-compatible directory (such as eStatement Manager CDA) and has following directory schema:



In the preceding example, the user "john" has two accounts with Boston Edison: one for gas and another for electricity, plus one account with ATT for phone service.

Choosing an Enrollment Model

Use the following guidelines to choose which enrollment model to use. You are not restricted to these enrollment models; you or Oracle Professional Services can implement a customized enrollment model.

Depending on how the user account is structured, choose the single-DDN or multiple-DDN model. If a single user can have multiple DDNs (accounts) per biller, then use the multiple-DDN model. If a single user can have only one DDN (account) per biller, then use the single-DDN model. However, since the multiple-DDN model also supports single-DDN, it is safe to use the multiple-DDN application for the single-DDN model. Use the multiple-DDN model even if the current requirement is for single-DDN, if multiple-DDN may be required in the future. Upgrading from single-DDN to multiple-DDN requires custom work because these two schemas are not compatible.

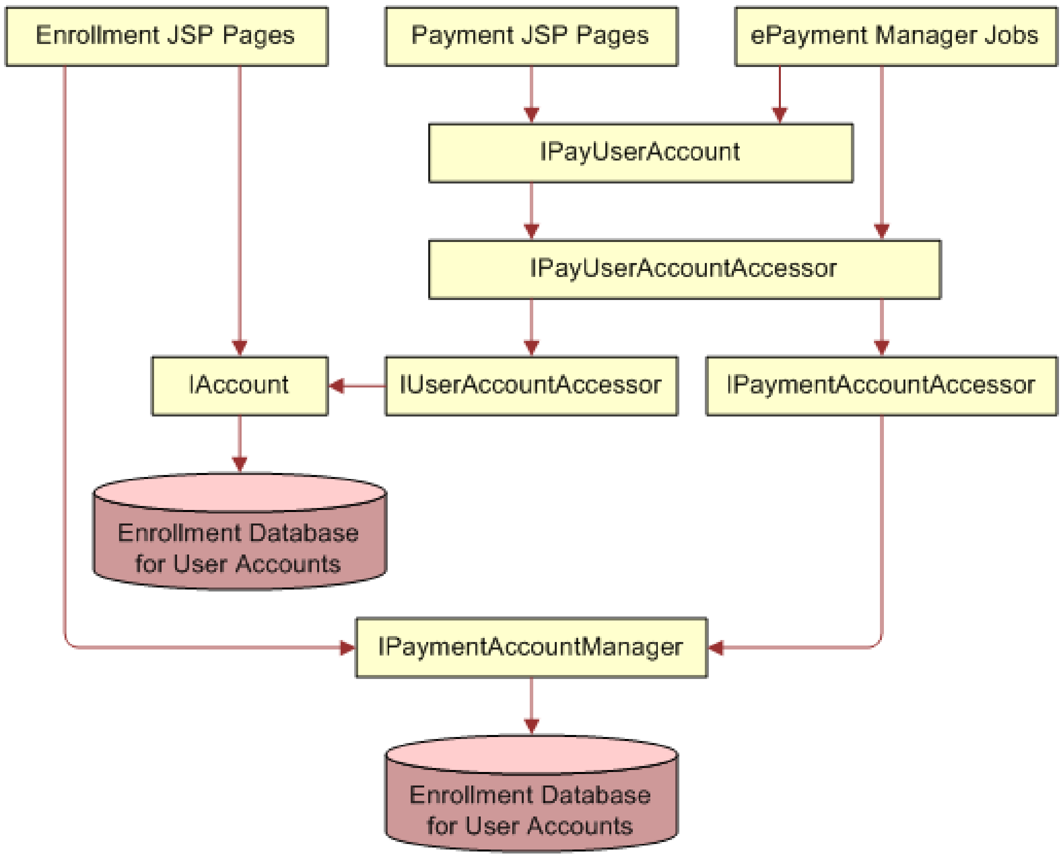
ePayment Manager Enrollment Architecture

ePayment Manager enrollment is considered as an independent part of ePayment, separate from the rest of the core ePayment Manager functionality, because of its complexity and flexibility.

The enrollment information required by ePayment Manager includes user accounts and payment accounts, which may come from different databases. ePayment Manager enrollment consists of the following enrollment functions:

- **Enrollment Management**: Adding, updating, and deleting ePayment Manager users, for example, enrolling a new user to ePayment Manager, adding a new bank account, etc. The ePayment manager JSP pages perform these functions, and are under *payment/user/jsp/enroll/user* in the WAR file. This function is not part of core ePayment Manager, but is important to the overall ePayment Manager application. Because the type of information required to manage enrolled users can vary greatly from deployment to deployment, this functionality is very flexible.
- **Enrollment Access**: Core ePayment Manager functionality, from the user interface to back end batch jobs, needs to access enrollment information, for example, to get a user email address or account number. Unlike the user enrollment information, ePayment Manager enrollment information is specific and well-defined.

The ePayment Manager enrollment architecture is shown in the following diagram:



Note that the enrollment database for User and ePayment Manager enrollment accounts may be the same database.

Three items are required to support enrollment:

- The front-end enrollment JSP pages
- The core ePayment Manager JSP pages
- The back-end ePayment Manager jobs

The ePayment Manager JSP pages are divided into two categories, the:

- **Enrollment management JSP pages**, under *payment/user/jsp/enroll* in the WAR file, add, update, and delete enrollment information
- **Core ePayment Manager JSP pages**, under *payment/user/jsp* in the WAR file, only reads enrollment information

To ensure maximum flexibility in managing enrollment, the enrollment JSP pages use `IAccount` from eStatement Manager to access user account information, and use `IPaymentAccountManager` to access payment account information. These pages are normally heavily customized during deployment in order to meet different enrollment requirements. The user accounts and payment accounts may or may not be in the same database.

However, since the core ePayment Manager JSP pages and the ePayment Manager engine both use the EJB interface `IAccount` from eStatement Manager to access user information, it does not matter where the user information is stored. It can be in eStatement Manager CDA table, a directory service accessible by JNDI, or in an external biller database. To minimize the impact of enrollment changes on the core ePayment Manager functionality, `IPayUserAccount` is defined as a wrapper class of `IAccount` and `IPaymentAccountManager`. Note that `IPayUserAccount` is a local object, not an EJB object. From `IPayUserAccount`, you can get both the user account information and the payment account information required for payment functionality. `IPayUserAccount` is also cached into the eStatement Manager `ISession` object after user login, so all the JSP pages can easily access it.

At the front end, the core ePayment Manager JSP pages only need to “view” enrollment information, so `IPayUserAccount` is sufficient for that purpose. At the back end, the payment jobs need to “view” enrollment information and to update payment account information. View and update is done through the adaptor interface, `IPaymentAccountAccessor`.

ePayment Manager Enrollment API

By default, we must use `IAccount` to access enrollment information. `IAccount` is very generic and difficult to use. By contrast, ePayment Manager enrollment information is very specific: we know we need account number, email address, check account number, etc. So ePayment Manager encapsulates `IAccount` to simplify and better store payment account enrollment data. The encapsulated `IAccount` also caches enrollment information in the session.

The following classes represent the enrollment models that ePayment Manager supports:

- **IPayUserAccount**: Defines a payment user enrollment. From this object, you can access all payment enrollment information: user info and payment account info. This interface represents uid node, DDN node and account number node. From this interface, you can walk down to DDN node and account number node. The essential information at the uid node is uid, password and email address. All other information is not required by ePayment Manager and is put into extended attributes of the object. For example, if you want to add user name to this node, it will be stored in extended attributes of `IPayUserAccount`.
- **IUserAccount**: Represents the information at the uid node. Each `IPayUserAccount` object includes one IUserAccount. However, you can't access `IUserAccount` from `IPayUserAccount` directly: `IPayUserAccount` has methods to help you access `IUserAccount` information.
- **BillerAccountInfo**: Represents DDN node. From this node, you can walk down to the account number node. The information required at this node is DDN name. Any other information is stored in extended attributes. For example, you can add a DDN description in the extended attributes.

- **AccountInfo**: Represents user account number node. The information required is account number and account description. Any other information is stored in extended attributes. For example, you can add an account name in the extended attributes.
- **Payment**: This is the main Servlet. Access to it requires login.
- **PaymentNoLogin**: This servlet does not require login. By default, you must login before you can enroll a payment account. But you can use `PaymentNoLogin` instead of `Payment` to enroll a payment account when first creating a new user login (For example, from “Enroll Now” in the sample application).
- **IPaymentAccountManager**: Accesses the *payment_accounts* table for the Hybrid enrollment model. In the Hybrid model, user info is accessed through `IAccount`.

For **Single-DDN**, there is no DDN node or account number node. So the DDN is passed in through URL, and the account number is part of the uid node. To support this with the Payment classes, a BillerAccountInfo object is created based on the DDN passed in from URL, and an AccountInfo object is created based on the account number that was saved into uid node.

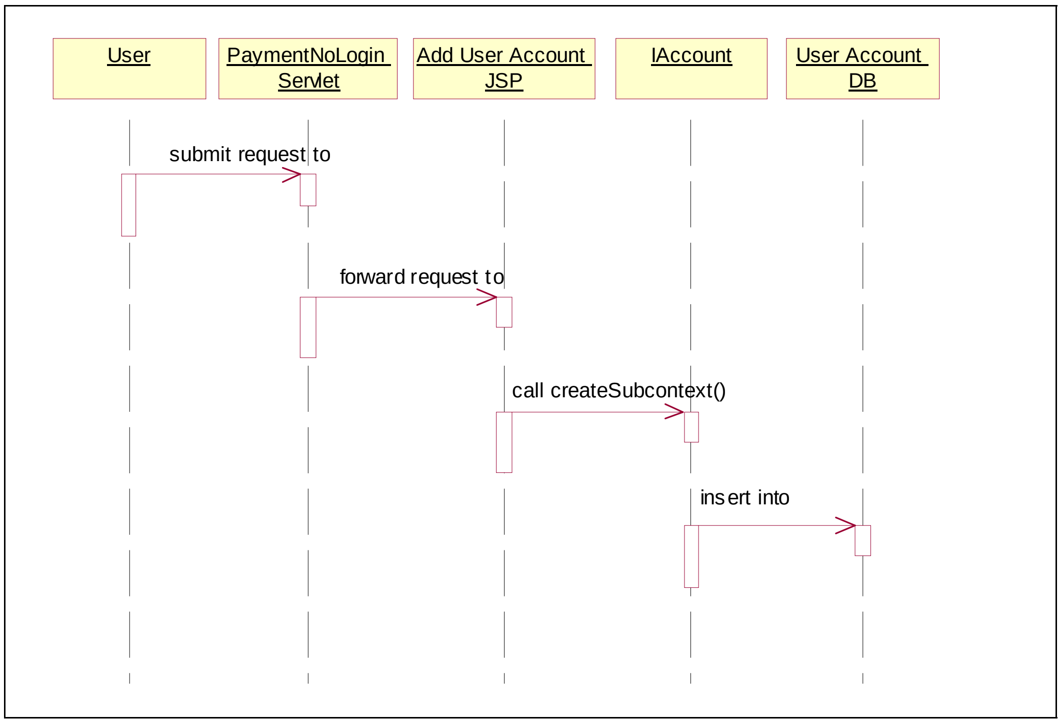
Major Enrollment Objects and Relationships

This section describes the major objects mentioned in the enrollment architecture.

IAccount

`IAccount` is used directly by the enrollment JSP pages to add, update and delete user information. The JSP pages also use the adaptor interface, `IUserAccountAccessor`, to construct an `IPayUserAccount` object. None of the core JSP pages and core ePayment Manager code use `IAccount`, which ensures maximum portability to different enrollment databases.

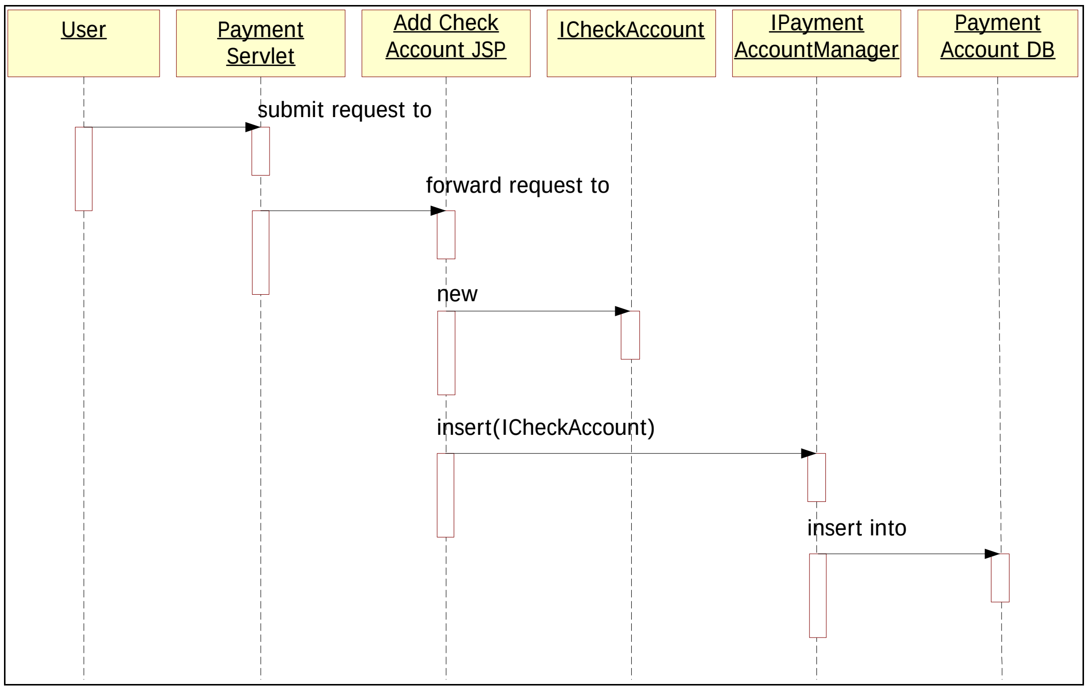
The following interaction graph illustrates how `IAccount` is used to enroll a new user account:



IPaymentAccountManager

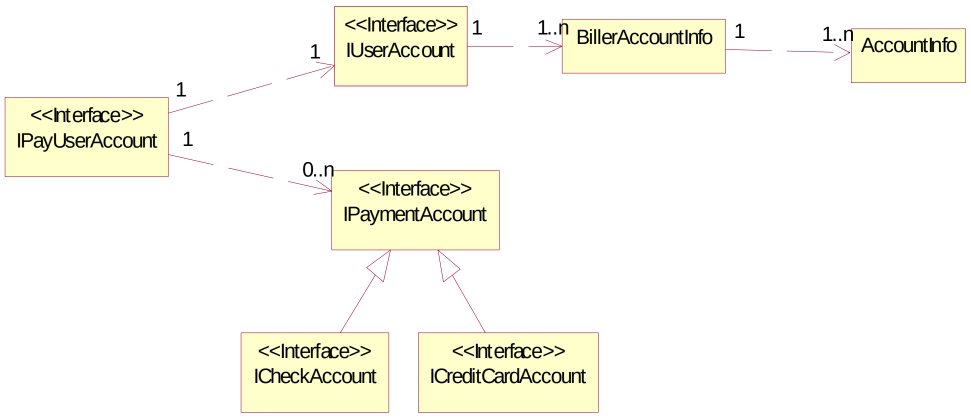
`IPaymentAccountManager` is an EJB, similar to `IAccount`, and is used directly by enrollment JSP pages to add, update, and delete payment account information. (Actually, it is used through the wrapper class `PaymentAccountManager`, see the ePayment Manager JavaDoc for more information). `IPaymentAccountManager` is also used by the adaptor interface, `IPaymentAccountAccessor`, to construct an `IPayUserAccount` object. None of the core JSP pages or core ePayment Manager code use `IPaymentAccountManager` directly, which ensures maximum portability to different enrollment databases.

The following interaction graph illustrates how `IPaymentAccountManager` is used to add a new check payment account:



IPayUserAccount

This interface represents an ePayment Manager user. It is a container class containing one `IUserAccount` object and a list of `IPaymentAccount` objects. The following diagram displays the object model for them:


<<Interface>> IUserAccount
BillerAccountInfo
AccountInfo
<<Interface>> IPayUserAccount
<<Interface>> IPaymentAccount
<<Interface>> ICheckAccount
<<Interface>> ICreditCardAccount
1
1..n
1
1..n
1
1
1
0..n

In the preceding diagram:

- **IUserAccount:** represents user account information.
- **BillerAccountInfo**: represents all the user's accounts with a biller. A user may enroll with multiple billers. In addition, for each biller, a user may enroll multiple accounts. This object includes a biller name, and a list of user accounts represented by an `AccountInfo` object.

- **AccountInfo**: Represents a single user account enrolled with a biller. It includes the account number and account description.

All the previously listed interfaces are local objects, and each of them is implemented by a class whose name is the same as the interface name omitting the leading letter "I". For example, `IPayUserAccount` is implemented by `PayUserAccount`. See the ePayment Manager JavaDoc for more details.

How the IPayUserAccount gets created and retrieved:

IPayUserAccount is the central object used in payment enrollment. It is easier to use than IAccount. It is also cached in ISession, which also means you can't get this object from ISession until you have logged in. There are two static functions defined in ePayment Manager (actually in its super class, but you should use ePayment Manager to access it because the super class will be removed in the future release).

Payment.getPayUserAccountFromSession() gets the IPayUserAccount object from the ISession. If it is not there (ISession has just logged in), ePayment Manager creates an IPayUserAccount from IAccount.

Payment.putPayUserAccountToSession() puts the IPayUserAccount object into ISession. You can use it to update the object in the session.

Since there are several different enrollment models, and accessing them requires different code, ePayment Manager provides an interface called IUserAccountAccessor to access user information, and IPaymentAccountAccessor to access payment account information.

Currently there are three implementation classes for IUserAccountAccessor:

- **JNDISingleDDNUserAccountAccessor** - Accesses user information through JNDI/CDA for the single DDN model.
- JNDIMultipleDDNUserAccountAccessor - Accesses user information through JNDI/CDA for the multiple DDN model.
- **EdocsUserAccountAccessor** - Accesses user information for the flat enrollment model.

The implementation is chosen by modifying *web.xml* and the payment settings.

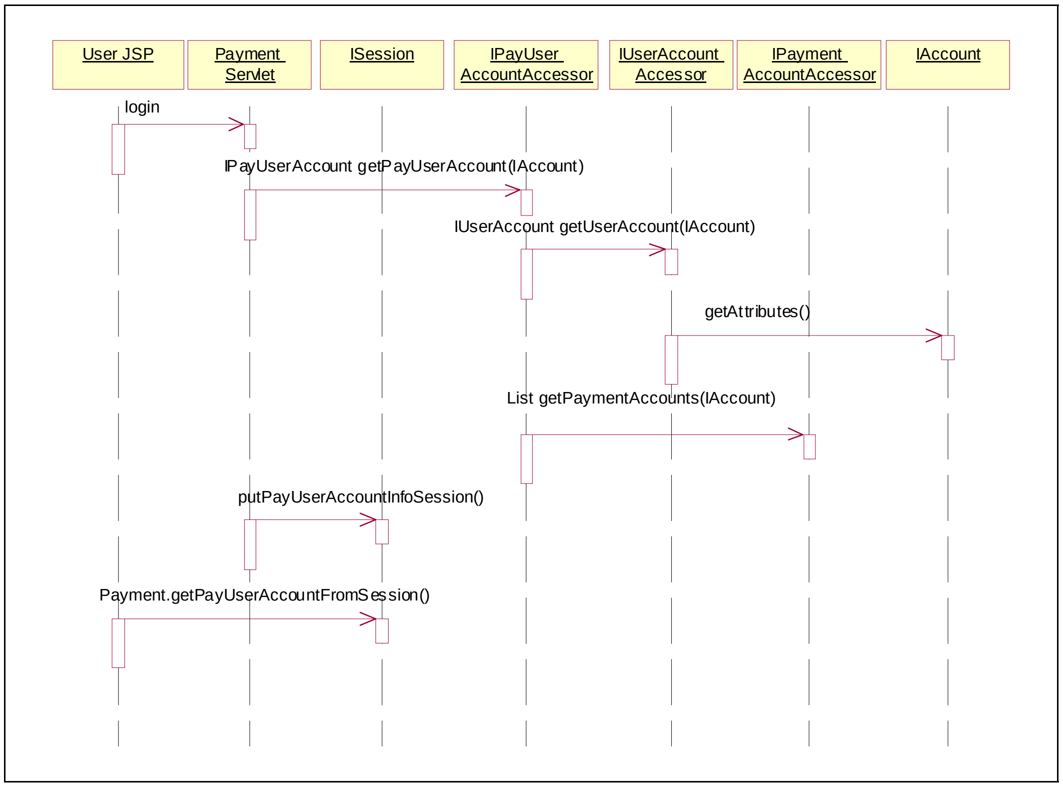
Currently there are three implementations of IPaymentAccountAccessor:

- **SSOPaymentAccountAccessor**- Accesses payment account information saved in the *payment_accounts* table for the Hybrid enrollment model.
- **JNDIPaymentAccountAccessor**- Accesses payment account information saved using JNDI/CDA for the Integrated enrollment model.
- **EdocsPaymentAccountAccessor**- Accesses payment account information saved using the flat enrollment model.

The implementation is chosen by modifying *web.xml* and the payment settings.

The implementation class of IPayUserAccountAccessor, PayUserAccountAccessor, reads *web.xml* or payment settings to get the correct implementation of IUserAccountAccessor and IPaymentAccountAccessor and the correct user and payment account information for different enrollment models.

The following interaction graph shows how IPayUserAccount is created:


User JSP
Payment Servlet
ISession
IPayUser AccountAccessor
IUserAccount Accessor
IPayment AccountAccessor
IAccount
login
IPayUserAccount getPayUserAccount(IAccount)
IUserAccount getUserAccount(IAccount)
getAttributes()
List getPaymentAccounts(IAccount)
putPayUserAccountInfoSession()
Payment.getPayUserAccountFromSession()

IUserAccountAccessor

This adaptor class hides the complexity of the **user account enrollment** databases. It does that by defining the necessary user account information required by ePayment Manager. All core ePayment Manager JSP pages and java code use this adaptor class to access user account information. You can re-implement this interface for your own enrollment database if the default implementation offered by ePayment Manager does not satisfy your requirements.

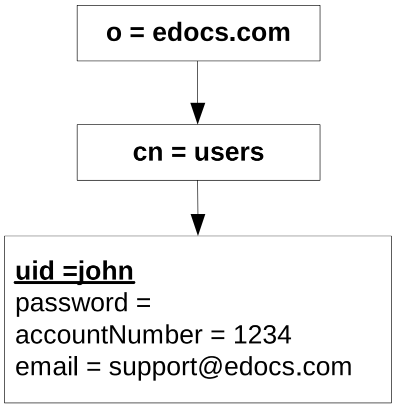
`IUserAccountAccessor` wraps `IAccount` to get the information required by ePayment Manager, such as account number and email address. The method you choose to retrieve that information depends on the directory schema of `IAccount`.

By default, ePayment Manager offers two implementations for `IUserAccountAccessor`:

- **JNDISingleDDNUserAccountAccessor**: Provides access to user information in JNDI/CDA for the single DDN model.
- **JNDIMultipleDDNUserAccountAccessor**: Provides access to user information through JNDI for the multiple DDN model.

You choose which implementation to use by modifying the Payment Settings in the Command Center.

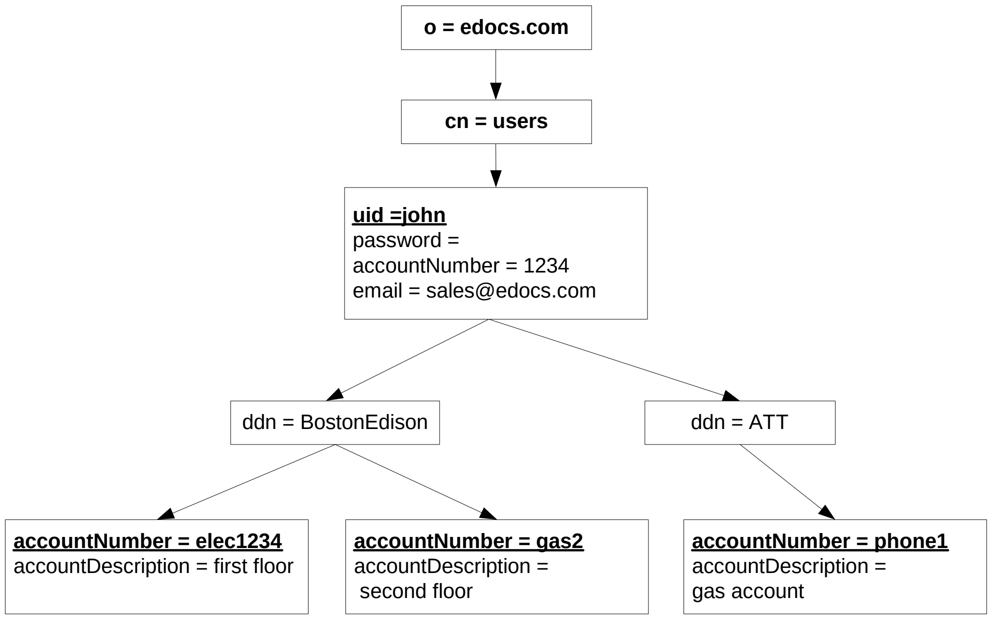
The following is an `IAccount` schema:



`IUserAccountAccessor` retrieves information from this schema using *com.edocs.payment.payenroll.usracct.JNDISingleDDNUserAccountAccessor*. When `IAccount`'s context is set to: "uid=..., cn=Users, o=edocs.com" node, `IUserAccountAccessor` should be able to get the account and email information from that node by calling `IAccount.getAttributes();`

If `IAccount` is implemented to support preceding schema, then you should be able to use `JNDISingleDDNUserAccountAccessor` without modification. It does not matter if the schema is based on CDA, LDAP or SSO.

The same rule applies to `JNDIMultipleDDNUserAccountAccessor`, which uses a schema similar to:



As long as your `IAccount implementation` supports a hierarchy similar to the preceding diagram, you can use the accessor class `JNDIMultipleDDNUserAccountAccessor` without modification.

IPaymentAccountAccessor

This adaptor class hides the complexity of **payment account enrollment** databases. It does that by defining the necessary payment account information required by ePayment Manager. All core ePayment Manager JSP pages and java code use this adaptor class to access user account information. You can re-implement this interface for your own enrollment database if the default implementation offered by ePayment Manager does not satisfy your requirements.

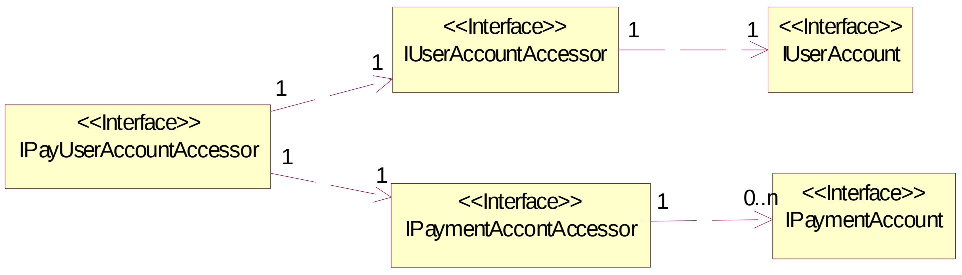
IPaymentAccountAccessor has one implementation class:

- **SSOPaymentAccountAccessor**: Provides access to payment account information saved in the *payment_accounts* table.

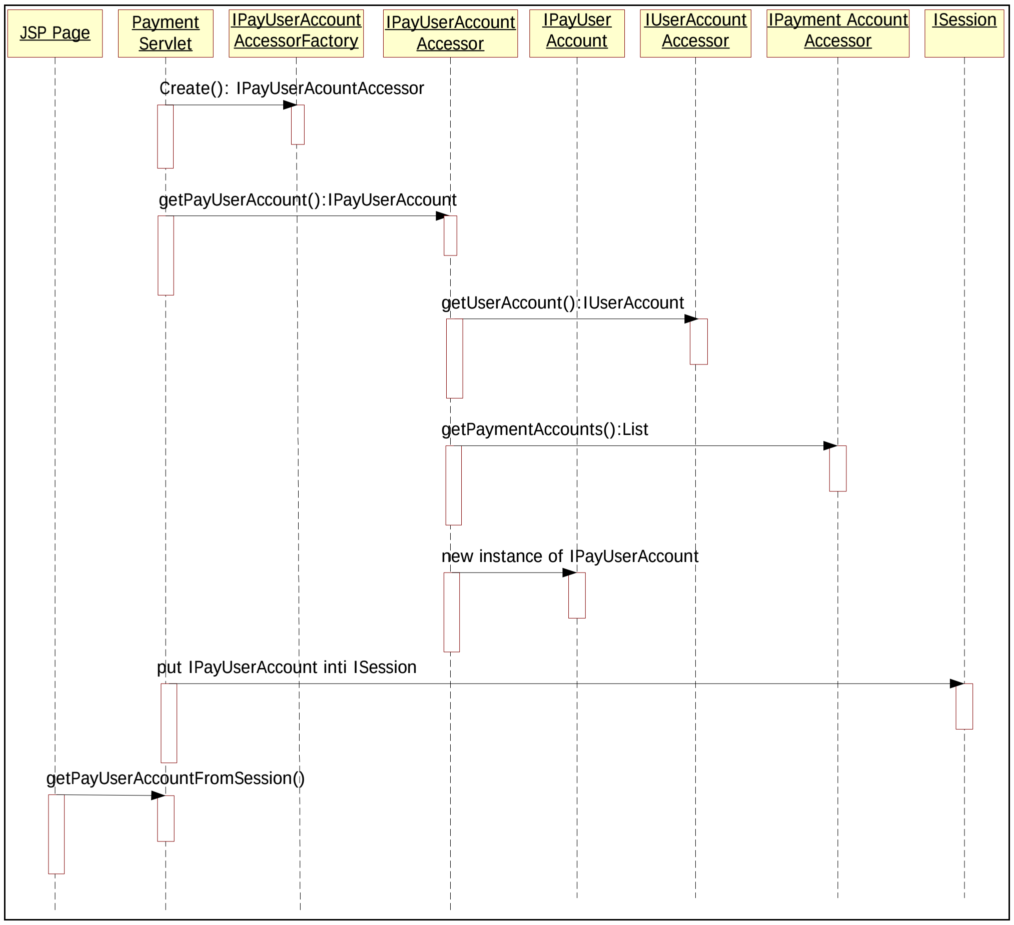
IPayUserAccountAccessor

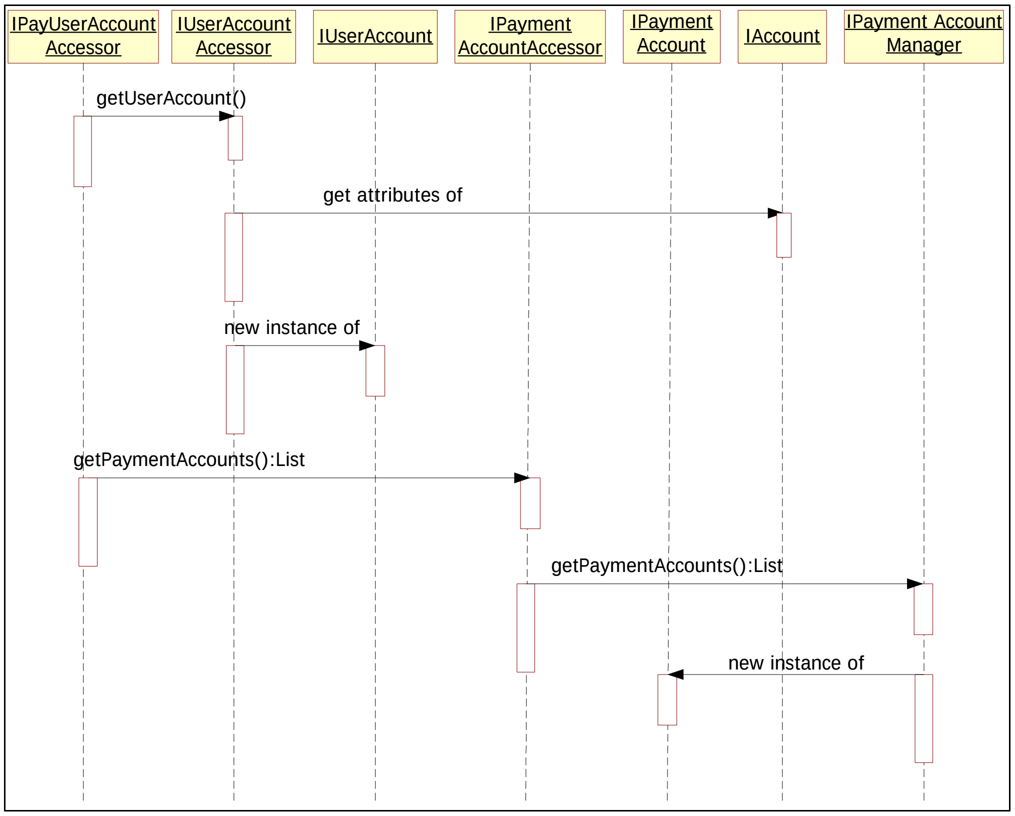
This class gets an IPayUserAccount object. It depends on IUserAccountAccessor to access user account information and IPaymentAccountAccessor to access payment account information.

The following diagram shows the object model for accessor objects:

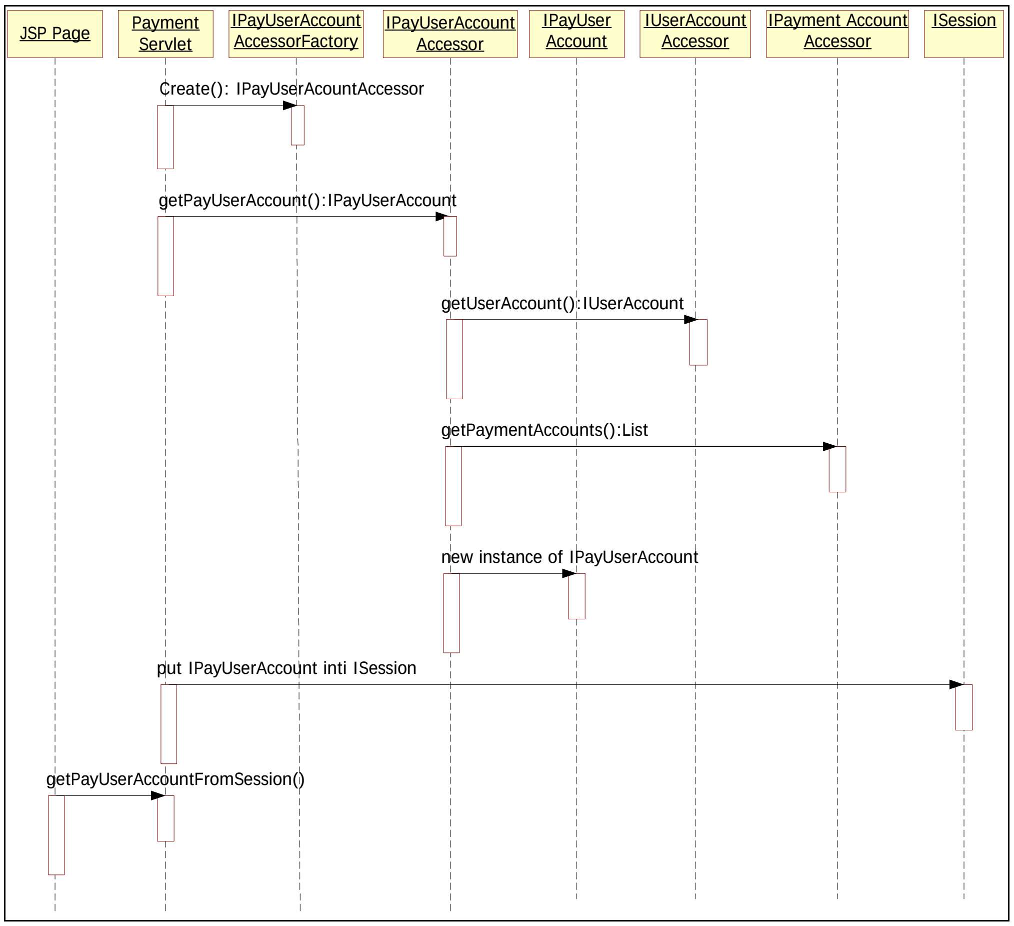


The following interaction graphs demonstrate how IPayUserAccount is created for ePayment Manager JSP pages:


JSP Page
Payment Servlet
IPayUserAccount AccessorFactory
IPayUserAccount Accessor
IPayUser Account
IUserAccount Accessor
IPayment Account Accessor
ISession
Create(): IPayUserAcountAccessor
getPayUserAccount():IPayUserAccount
getUserAccount():IUserAccount
getPaymentAccounts():List
new instance of IPayUserAccount
put IPayUserAccount inti ISession
getPayUserAccountFromSession()



The following diagram shows how `IPayUserAccount` is saved into the `ISession` object, and can be used by all ePayment Manager JSP pages (including enrollment JSPs).


JSP Page
Payment Servlet
IPayUserAccount AccessorFactory
IPayUserAccount Accessor
IPayUser Account
IUserAccount Accessor
IPayment Account Accessor
ISession
Create(): IPayUserAcountAccessor
getPayUserAccount():IPayUserAccount
getUserAccount():IUserAccount
getPaymentAccounts():List
new instance of IPayUserAccount
put IPayUserAccount inti ISession
getPayUserAccountFromSession()

Payment Servlet

The `Payment` servlet defines two static functions:

- *Payment.getPayUserAccountFromSession()* gets `IPayUserAccount` from eStatement Manager's `ISession`. If `IPayUserAccount` is not there (which is the case if you just logged in), Payment creates an `IPayUserAccount` from `IAccount`.
- *Payment.putPayUserAccountToSession()* puts `IPayUserAccount` into `ISession`. You can use this method to update the object in the session.

Customizing ePayment Manager Enrollment

eStatement Manager provides two sample applications that use different user enrollment methods: Sample and UMFSample. Sample uses CDA to store user information in a user directory schema based on LDAP.

See the *SDK Guide for Oracle Siebel eStatement Manager* for more information about the eStatement Manager sample applications and how to implement and modify them.

- ePayment Manager provides the paymentComplex sample application, which shows how to support multiple DDNs per user. This sample application supports the multiple-DDN model; it doesn't correspond to any current eStatement Manager samples.

The Payment Complex application requires that the user login (pass authentication) before accessing payment functions. The login is defined by the eStatement Manager UMF implementation.

Customizing ePayment Manager Enrollment JSP Pages

All ePayment Manager enrollment is done through JSP pages. The servlet only forwards enrollment requests to the JSP pages. The URL looks something like:

```
Payment?app=Payment&form=forward&jsp=jsp_name
```

- The URL instructs the servlet to forward this http request to `jsp_name` after the user passes authentication.
- This method requires the user to login in order to access this JSP page. If you don't want to login, replace `Payment` with `PaymentNoLogin`.

The following table lists the payment enrollment JSP pages, which are all under *<WebRoot>/payment/user/jsp/enroll*. There are two sets of JSPs, one for user account info and another for payment account info.

JSP Name	**Description**
User JSP for multiple DDN in *user/multiple_ddn*	All pages under this directory are for multiple-DDN enrollment model only. They are the pages for enrolling a user account.
user_enroll.jsp	Enroll a new multiple-DDN user.
user_enroll_summary.jsp	Summary of user enrollment information, includes both user account and payment accounts.
add_update_delete_biller _account.jsp	Add/Update/Delete a user's account with a biller.
UserLogin.jsp	User login page for multiple-DDN.
UserLogout.jsp	User logout page for multiple-DDN.
Payment Account JSP	The payment accounts are saved into the *payment_accounts* table of the ePayment Manager database and being accessed by `IPaymentAccountManager`.
add_update_ccard.jsp	Add/Update a credit card account.
delete_payment_account.jsp	Delete a check/credit card account.
add_update_check.jsp	Add/Update a check account.

The following table provides special notes about each type of payment account during enrollment:

Account	**Note**
check account	When a check account is created, if you decide to use an ACH prenote to validate that account, you can choose whether or not to associate a DDN with that check account. If there is no DDN, then any DDN's pmtSubmitEnroll job picks up this check account. Otherwise, the check account will only be processed by the pmtSubmitEnroll job with the matching DDN.
credit card account	When a credit card account is created or updated, you can decide whether to authorize this credit card during enrollment time. You must choose a DDN which is configured with a Credit Card cartridge to use authorization. The authorization amount is $1, which will not be settled (it has a status of "auth-only"). It will be eventually be discarded by the payment processor.

Changing Web Context

The default Web context is paymentComplex for multiple DDN. To change the Web context, update *application.xml* in *ear-payment-complex.ear*, depending on which application you plan to use.

Enrolling payment account and user account number (for multiple DDNs) when enrolling a new user

Using the default JSP pages, when enrolling a new user, you won't be able to enroll a payment account. In the multiple-DDN case, you won't be able to enroll the user account either. To allow payment accounts to enroll during user enrollment time, use the create payment account URL and create user account URL in the "Edit Profile" page. Just replace the "app=Payment" with "app=PaymentNoLogin". This way, creating a payment account and user account number won't require login.

To add custom fields to user account information

Sometimes you want to add and retrieve more information to and from the enrollment database. Since this information is not essential to ePayment Manager, it is stored in the extended attributes of `IUserAccount`, `BillerAccountInfo` or `AccountInfo`.

The default implementation of `IAccount` is based on the eStatement Manager CDA database and we need to define a schema for each enrollment model (see the preceding enrollment models for single-DDN and multiple-DDN). To add a new field to one of the uid, DDN or accountNumber nodes, treat it as an extended attribute of `IUserAccount`, `BillerAccountInfo` or `AccountInfo`, respectively.

For example, to add "first name" as part of the uid attributes. The following steps describe how it is implemented. You can create your own new attribute by following similar steps.

1. Add a "firstname" attribute to the CDA schema table, as described in the *SDK: Implementing a User Management Framework* document.

2 Edit *payment/user/jsp/enroll/user/<multiple_ddn* or *single_ddn>/user_enroll.jsp* and add "First Name" as a table element, using the name attribute *firstname* on the HTML `input` element.

```
<input type="text" name="firstname" value="">
```

3 Open *payment/user/jsp/enroll/user/user_enroll.inc*, and find the `insertUserInfoCDA()` method. This method actually already does the work for you: extract "firstname" from request and insert it into CDA by using `IAccount`. It also updates the `IPayUserAccount` to add this new extended attribute. The updated `IPayUserAccount` will be put into `ISession`.

4 Also, find the `updateUserInfoCDA()` method in *user_enroll.inc*. This method actually already does the work for you: extract "firstname" from request and update it into CDA by using `IAccount`. You should also update the `IPayUserAccount` with this new extended attribute. The updated `IPayUserAccount` will be put into `ISession`.

5 In other JSP pages, you should be able to get `IPayUserAccount` from `ISession`, which includes the new extended attribute:

```
IPayUserAccount ipua =
Payment.getPayUserAccountFromSession(…);

String firstName=ipua.getExtendedAttribute("firstname");
```

6 Note that the `IPayUserAccount` object retrieved by the back end (Command Center) jobs also includes the extended attributes.

Adding custom fields to check or credit card accounts

You cannot add an arbitrary number of custom fields to a check or credit card account. Instead, ePayment Manager provides a number of flexible fields for this purpose.

For each payment account, there are two flexible string fields and one flexible date field that can be used for customization. These custom fields are accessed as extended attributes of the `ICheckAccount` and `ICreditCardAccount` objects.

In the *payment_accounts* table, these three fields, *flex_field_1*, *flex_field_2* and *flex_date_1* are the extended attributes. The name of the attribute must be the name of the flexible field column in the *payment_accounts* table, such as *flex_field_1*, *flex_field_2* and *flex_date_1*.

For example, you can use *flex_field_1* as the name of the financial institution:

To insert it:

```
IPaymentAccount pa = new PaymentAccount();

ps.addExtendedAttribtue("flex_field_1", "Fleet Boston
Financial Corp");

…//call set methods to set other fields

PaymentAccountManager pam = new PaymentAccountManager();
```

```
pam.insert(pa);
```

To retrieve it:

```
IPayUserAccount pua = Payment.getPayUserAccountFromSession(…);
String bankName = pua.getExtendedAttribute(“flex_field_1”);
```

Integrating ePayment Manager with Other User Account Enrollment Databases

Sometimes user account information comes from an existing customer database instead of from JNDI/CDA. In this case, you need to re-implement `IAccount`. You may also need to re-implement the `IUserAccountAccessor` interface. In this case, the payment accounts are still saved into *payment_accounts* table of ePayment Manager database.

If your `IAccount` implementation has a similar enrollment schema as JNDISingleDDNUserAccountAccessor or JNDIMultipleDDNUserAccountAccessor, the default implementation may be able to work with `IAccount` to get the information it needs. In this case, there is no need for you to re-implement `IUserAccountAccessor`. Otherwise, you must re-implement it. To implement `IUserAccountAccessor`:

1. Implement the eStatement Manager interface IAccount first. See the *SDK Guide for Oracle Siebel eStatement Manager* to learn how to implement IAccount. Make sure you implement all the methods of IAccount.

2. Write your own implementation class of IUserAccountAccessor. You should extend your class from the UserAccountAccessorImpl class. Overwrite all the methods you inherited to get user information from the database. You must implement all three methods.

3. Package your class into payment_custom.jar of each ePayment Manager ear file. For details about how to do that, see the *Deploying and Customizing J2EE Applications Guide for Oracle Siebel eStatement Manager*.

4. Change user_account_accessor in web.xml to point your implementation class, and change the IUserAccountAccessor field in Payment Settings to point to your implementation class.

5. Customize the enrollment JSP pages, if necessary. If IAccount is implemented properly and has a schema similar to the default ones, you may not need to customize the JSP pages. Otherwise, you must customize the JSP pages to make them work properly with your new enrollment database.

6. Re-deploy the EAR files.

Integrating ePayment Manager with Other Payment Account Enrollment Databases

We strongly recommend that all payment accounts be saved into the *payment_accounts* table of ePayment Manager database. This default implementation should be sufficient to meet the needs of most deployments.

However, in rare cases, payment account information may come from an existing customer database, using the following steps:

1. Re-implement the `IPaymentAccountAccessor` interface to communicate with the custom payment account database. Extend your class from the `PaymentAccountAccessorImpl` class. Overwrite the methods you inherited to get payment account information from the database.

2. Package your class into *payment_custom.jar* of each ePayment Manager EAR file. For more information about how to do that, see the *Deploying and Customizing J2EE Applications Guide for Oracle Siebel eStatement Manager*.

3. Change *payment_account_accessor* in *web.xml* to point your implementation class, and change the `IPaymentAccountAccessor` field in Payment Settings to point to your implementation class.

4. Customize the payment account enrollment JSP pages to use the new database to add, update, and delete payment accounts. Do not use `IPaymentAccountManager`, because it uses the ePayment Manager database.

5. Re-deploy the EAR files.

4 Recurring Payments

Overview

ePayment Manager's recurring payment feature is a very complicated process that involves a great deal of business logic. This section discusses the recurring payment processing in detail.

Recurring payments consist of actions at the front-end (UI) and back end (Command Center jobs). The UI allows a user to insert/update/delete a recurring payment, and the back end pmtRecurPayment job actually makes the payment.

To understand how recurring payment works, we need to track the changes to the information in the *recurring_payments* table:

Column Name	**Comment**
AMOUNT_TYPE and AMOUNT	These two columns record how the payment amount is generated. They are only updated through the UI and are used by back-end jobs to calculate how much to pay. The valid values of AMOUNT_TYPE are: • "fixed amount": pay a fixed amount and the amount value is specified by AMOUNT column. • "amount due": pay amount due on the bill and, AMOUNT column is not used (null). • "minimal due": pay minimuml amount due on the bill and AMOUNT column is not used (null). • "less due": means pay the amount due if it is less than the value of the AMOUNT column; otherwise, pay nothing and send email notification. • "upto amount": payD the amount due if it is less than the value of the AMOUNT column; otherwise, pay the value of AMOUNT and send email notification.

Column Name	**Comment**
PAY_INTERVAL DAY_OF_PAY_INTERVAL MONTH_OF_PAY_INTERVAL	These three columns record how the payment date is generated. They are only updated through the UI, and are used by back-end jobs to calculate when to pay. The valid values of PAY_INTERVAL are: • "weekly": user specified to make payments weekly. The day of week is specified by DAY_OF_PAY_INTERVAL. The MONTH_OF_PAY_INTERVAL is irrelevant. • "monthly": user specified to make payments monthly. The day of month is specified by DAY_OF_PAY_INTERVAL. The MONTH_OF_PAY_INTERVAL is irrelevant. • "quarterly": user specified to make payments quarterly. The day of month is specified by DAY_OF_PAY_INTERVAL. The month of quarter is specified by MONTH_OF_PAY_INTERVAL (one of 1,2 or 3) .
START_DATE END_DATE CURR_NUM_PAYMENTS MAX_NUM_PAYMENTS STATUS	These four columns determine when to start the recurring payment and when to stop it. START_DATE, END_DATE and MAX_NUM_PAYMENTS can only be updated through the UI. START_DATE is required, but you set only one of the END_DATE (end by that date) or MAX_NUM_PAYMENTS (end when this number of payments is made). The recurring payment STATUS is "active" when it is created and it has not reached either END_DATE or MAX_NUM_PAYMENTS. When one of them is reached, the STATUS is changed to "inactive" and the recurring payment will never take effect again. If END_DATE is chosen, NEXT_PAY_DATE (the pay date for the next bill needs to be paid) is >= START_DATE and <= END_DATE, the bill will be paid. The STATUS is set to inactive if NEXT_PAY_DATE > END_DATE. If MAX_NUM_PAYMENTS is chosen, the STATUS is changed to inactive when CURR_NUM_PAYMENTS reaches MAX_NUM_PAYMENTS.
LAST_PAY_DATE	This is the pay date of last bill. It is set to 01/07/1970 when recurring payment is created to indicate that there is valid information.
NEXT_PAY_DATE	This is the pay date of next bill. When the recurring payment job runs, it schedules a payment with a pay date of NEXT_PAY_DATE. Note: NEXT_PAY_DATE is calculated based on LAST_PAY_DATE and PAY_INTERNAL. For details, see below.

Column Name	**Comment**
LAST_PROCESS_TIME	Records the date of the last time the recurring payment job ran. During the recurring job synchronization process, ePayment Manager retrieves bills between LAST_PROCESS_TIME and current time to avoid retrieving duplicated bills and so, improve the performance. Note: Previous versions of ePayment Manager recorded both date and time information in LAST_PROCESS_TIME. It was found that a bill could be lost if the bill was indexed the second time on the same date because the DOC_DATEs of bills don't include time information. Currently, the LAST_PROCESS_TIME only includes date info.
BILL_SCHEDULED BILL_ID	BILL_SCHEDULED indicates whether the latest bill identified by BILL_ID has been paid or not. BILL_ID decides whether a recurring payment needs to synchronize with eStatement Manager.
PAYER_ACCT_NUM	If multiple biller accounts are not present, this field is used to store the biller account number. If multiple biller accounts are present, this field is populated with an ID generated from the *PAYMENT_RECURRING_ID_SEQ* sequence.
PAYEE_BANK_ACCOUNT_ID	For check payments, this field is used to store the ID of the payee bank account.

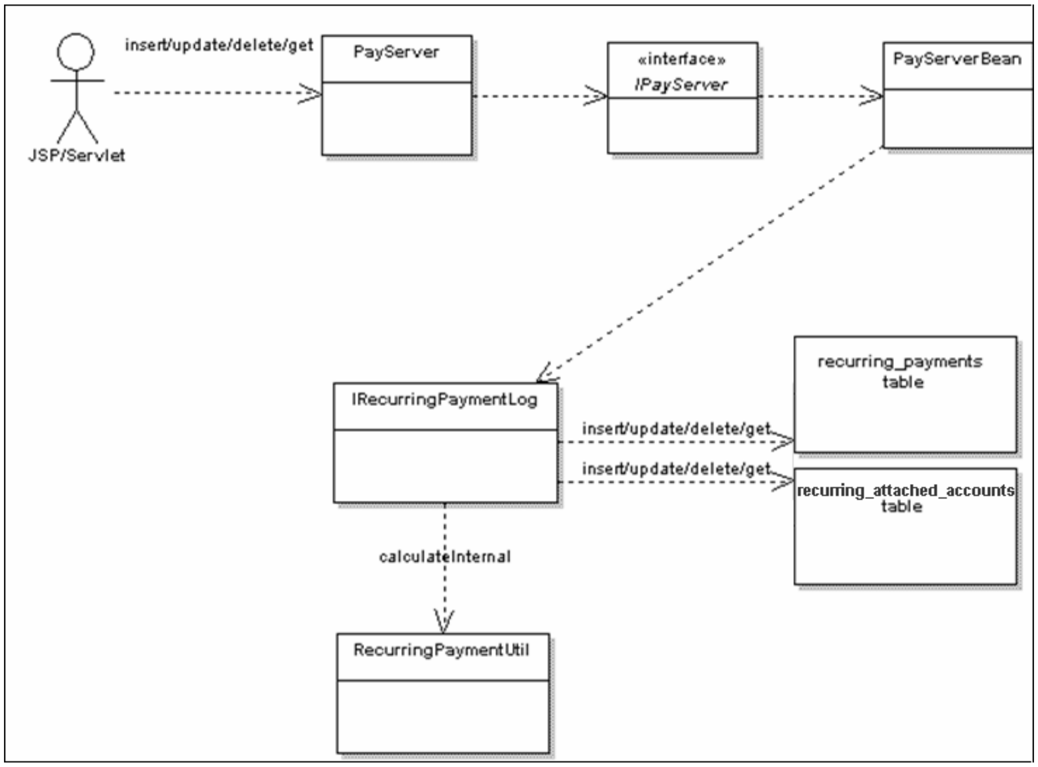
When multiple biller accounts are present RECURRING_ATTACHED_ACCOUNTS table is populated with the biller accounts (attached accounts). This table consists of five fields as follows.

PAYER_ID	Equals to the PAYER_ID of the corresponding record in RECURRING_PAYMENTS table.
PAYEE_ID	Equals to the PAYEE_ID of the corresponding record in RECURRING_PAYMENTS table.
RECURRING_ID	Equals to the PAYER_ACCT_NUM of the corresponding record in RECURRING_PAYMENTS table.
ACCOUNT_NUMBER	The biller account number.
INDEXER_DDN	Name of the corresponding indexer DDN.

Recurring Payment UI

The UI sets up a recurring payment. The UI allows you to insert, update, and delete a recurring payment and get back the list of recurring payments.

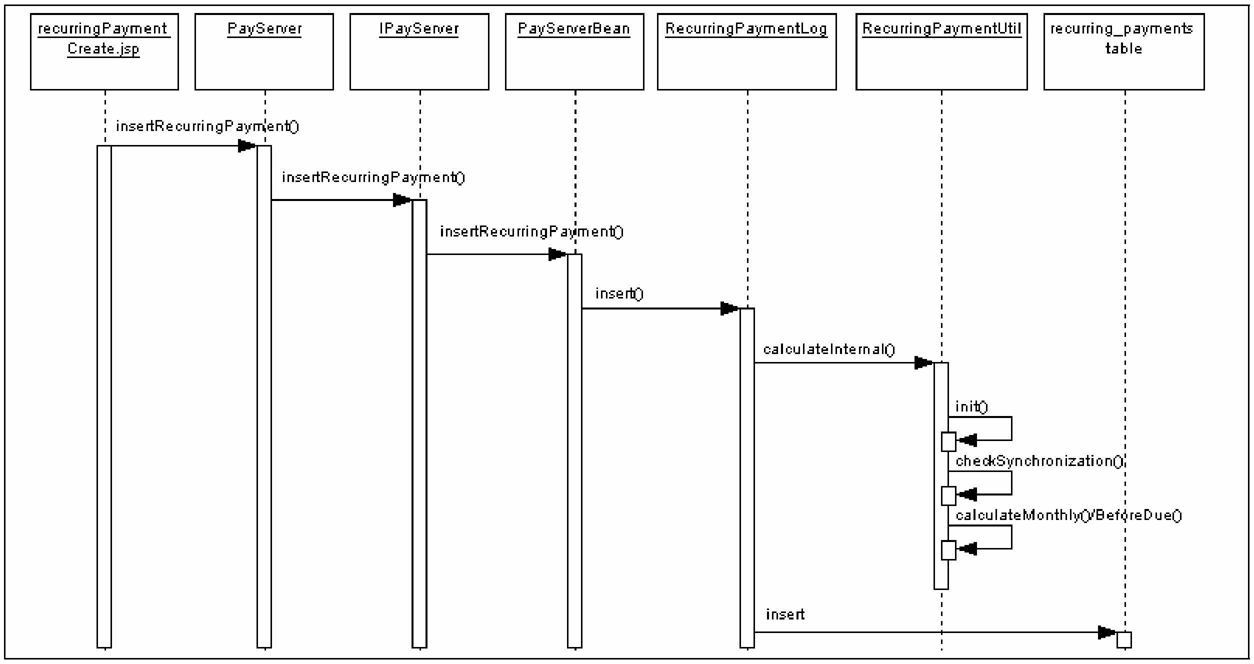
The following UML diagram illustrates the objects involved in the process:



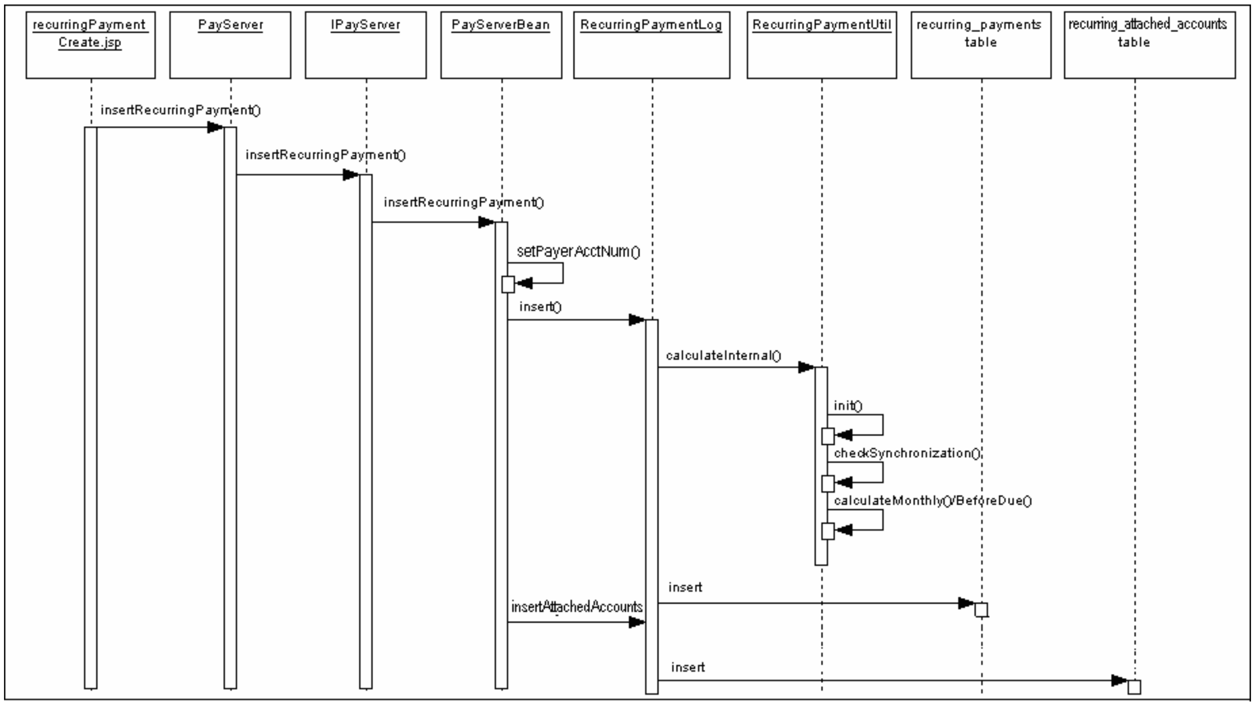
Retrieving and deleting recurring payments from the database is straightforward, so the next sections discuss what happens when a recurring payment is inserted or updated.

Insert Recurring Payment

The following sequence diagram demonstrates what happens when a recurring payment is inserted into database using the UI in B2C scenario. In B2C scenario, a recurring payment can have only one biller account):



In B2B scenario a recurring payment can have one or more biller accounts (attached accounts).The sequence diagram for inserting a recurring payment from UI in B2B scenario is as follows:



The next section explains RecurringPaymentUtil.calculateInternal(). This method calculates the next_pay_date and status of the recurring payment before it is being inserted into database.

This method calculates the internal states of recurring payment differently for insert and update. For the insert operation, this method does these things:

1. Call init() method: This method sets some of the recurring payment fields.

- If user chooses to end recurring payment by maximum number of payments, set end_date to 01/01/3000 00:00:00.
- If user chooses to end recurring payments by a fixed date, set max_num_payments it java.lang.Integer.MAX_VALUE.
- Set last_pay_date to 01/01/1970 00:00:00; this means no bill has been paid.
- Set bill_scheduled to "Y" if the recurring payment is fixed amount and fixed date. Note: In this case, the flag should be always true because whenever a payment is made, the next payment is calculated. It has the same effect as making the next bill available immediately.
- Set last_process_time to start_date, which by default must be tomorrow or later. This means that any bills indexed through today (inclusive) won't be picked up by recurring payment.

Note: The recurring payment UI checks whether there are unpaid bills when a recurring payment is setup, and reminds the user to make a one-time payment to pay the outstanding bill.

2. Call the `checkSynchornization` method: Checks whether any required information is missing from recurring payment before inserting it into the database.

3. Check whether the recurring payment has expired by checking the current number of payments against maximum number of payments. Note: This check always return false for insert case.

4. Calculate the *next_pay_date* by calling one of `calculateMonthly()`, `calculateQuarterly()`, `calculateWeekly()` or `calculateBeforeDue()` depending on whether *pay_interval* is "monthly", "quarterly" or "weekly" or "before_due" respectively.

- Call `calculateMonthly()` when pay_interval is "monthly"

  This method calculates the next pay date, which is based on last_pay_date, start_date and day_of_pay_internal. Since last_pay_date is 01/01/1970, the next_pay_date is the nearest date with day_of_pay_internal after the start_date. If date_of_pay_internal is 29, 30 or 31 and there is no such date in that month, the last day of that month is used. After next_pay_date is calculated, it is checked against the end_date. If next_pay_date passes the end_date, the status of the recurring payment is set to "inactive".

  The following table displays some examples of how next_pay_date is calculated:

Day_of_pay_interval	**Start_date**	**Next_pay_date**
1	Sep 10	Oct 1
10	Sep 10	Sep 10
15	Sep 10	Oct 15
31	Sep 10	Sep 30

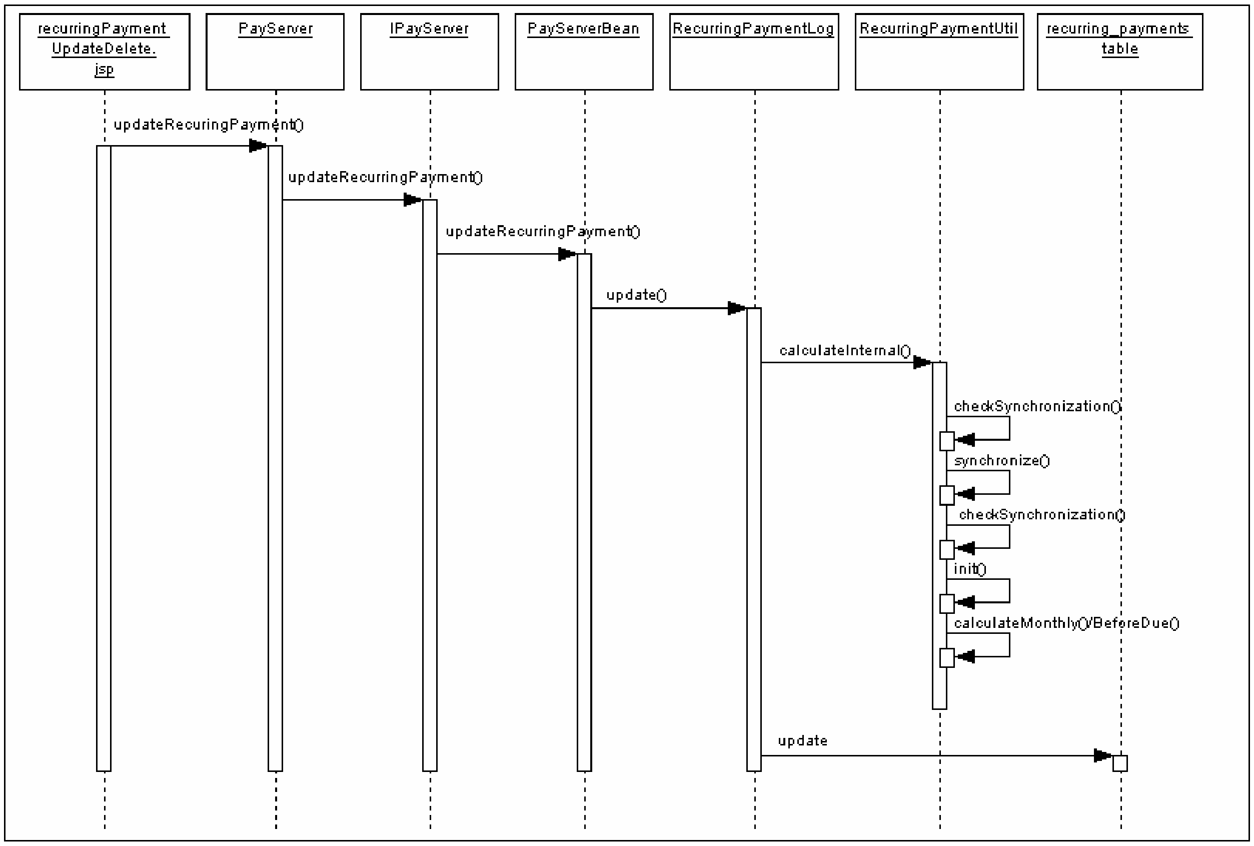
- Call `calculateQuarterly()` when *pay_interval* is "quarterly": works similar to "monthly"
- Call `calculateWeekly()` when *pay_interval* is "weekly": works similar to "weekly".

- Call `calculateBeforeDue()` when *pay_interval* is “before due”: since there is no bill yet (bill due date is null), the recurring payment status is set to "active" and the *next_pay_date* is set to 01/01/3000.

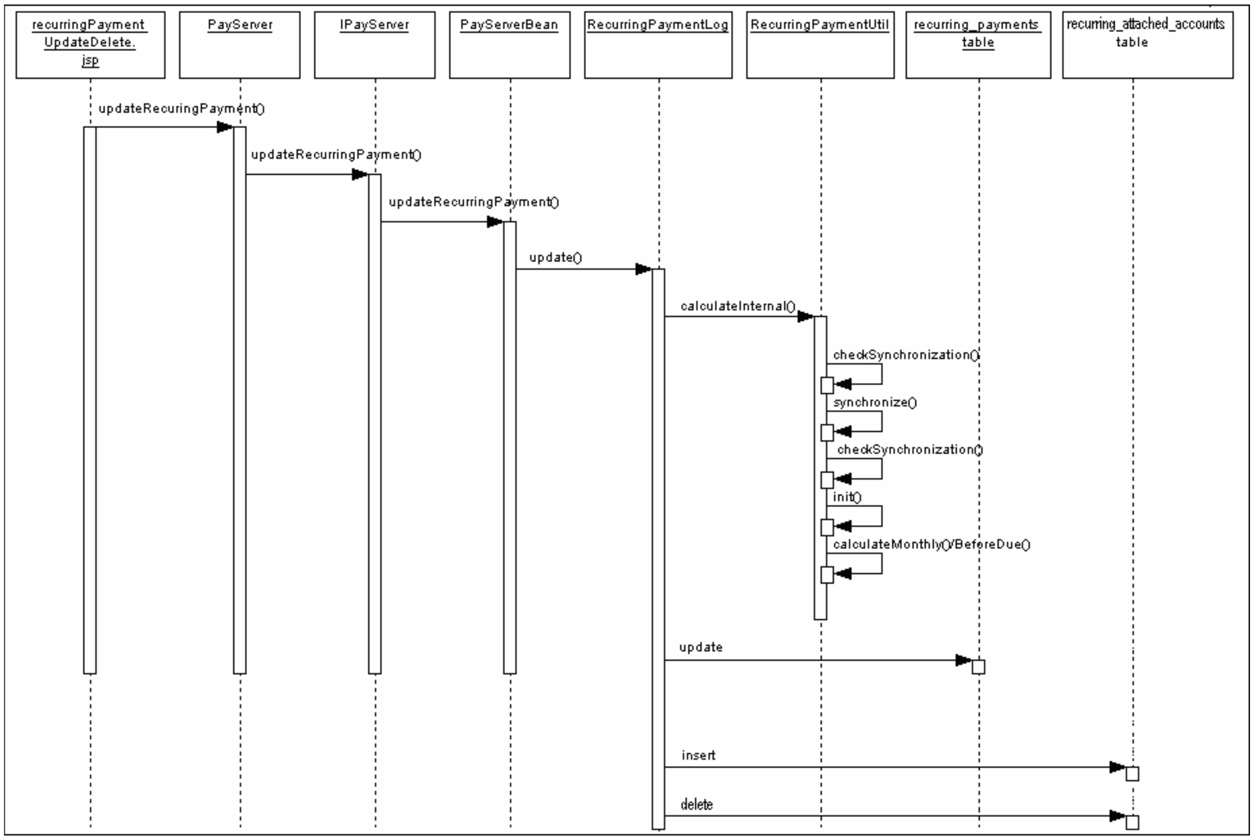
Update Recurring Payment

This section assumes that the UI prevents a user from updating a recurring payment from fixed date to before due date or vise versa. If the UI is changed to allow a user to do so, the behavior of recurring payment is not tested.

The following sequence diagram demonstrates what happens when a recurring payment is updated using the UI into the database in B2C scenario.



In B2B scenario, the sequence diagram is as follows.



The next section explains `RecurringPaymentUtil.calculateInternal()`. This method calculates the *next_pay_date* and the status of the recurring payment before it is inserted into database. This example starts from `IRecurringPaymentLog.update()`. Note that this method is also used for update by the back end job.

1. Call IRecurringPaymentLog.update()

2. Call `RecurringPaymentUtil.calculateInternal()`

3. Call `checkSynchronization()` method to check whether the information required for recurring payment is present.

4. If `checkSynchronization()` throws an exception indicating missed information, then:

- Call `synchronize()` method to read the missed information from the database and populate the missing information into the recurring payment object.
- Call `checkSynchronization()` again to make sure the required information has been populated.
- Call `init()` method: unlike the insert operation, this method checks whether the recurring payment has started or not by checking the *last_pay_date* (01/01/1970 means not started yet) and then sets the last process time to the *start_date* of the recurring payment if the recurring payment has not been started. The last process time won’t be updated if recurring payment has been started.

5. Check whether the recurring payment has expired by checking the current number of payments against maximum number of payments. If true, set the recurring payment as inactive and return.

6. Calculate *next_pay_date* and recurring payment status by calling one of `calculateMonthly()`, `calculateQuarterly()` or `calculateWeekly()` based on *pay_interval* of "monthly", "quarterly" or "weekly".

- Call `calculateMonthly()` when *pay_interval* is "monthly", to calculate the next pay date.

  If the *last_pay_date* is 01/01/1970, then the *next_pay_date* is calculated based on the *start_date* and *day_of_pay_interval*. It is set to the nearest date with *day_of_pay_interval* as day of month after the *start_date*. This is the same as the insert case. See previous section for details.

  If the *last_pay_date* is not 01/01/1970, that means that recurring payment has started, so the *next_pay_date* is calculated based on the *last_pay_date* and *day_of_pay_interval*. It is set to the date one month after the *last_pay_date*. Note: Here, the calculation doesn't depend on the current date. For example, if the recurring payment job runs today on Oct 1, the *last_pay_date* is Aug 30 and *day_of_pay_interva*l is 30, the *next_pay_date* will be Sep 30 (not Oct 30 as you may think) even though this date is in the past. In the case of fixed date and pay amount due, this can pose a problem if there is no bill for a certain month: the pay date will be in the past. To fix the problem, the recurring payment job will move the *last_pay_date* ahead by one month if there is no bill for that month. See following discussion for more details about the recurring payment job.

  If *day_of_pay_interval* is 29, 30 or 31 and there is no such date in that month, the last day of that month is used.

  After *next_pay_date* is calculated, it is checked against the *end_date* and if it passes the *end_date*, the status of the recurring payment is set to "inactive".

- Call `calculateQuarterly()` when *pay_interval* is "quarterly": works similar to "monthly"

- Call `calculateWeekly()` when *pay_interval* is "weekly": works similar to "weekly".

- Call `calculateBeforeDue()` when *pay_interval* is "before_due":

  First, check whether the recurring payment has been synchronized (bill due date not null) and if so, set status to active and next pays date to 01/01/3000 and return.

  Calculate the proposed next pay date by current bill due date and *day_of_internal*.

  If the proposed *next_pay_date* is before *start_date*, set the status of recurring payment to "active" and *next_pay_date* to 01/01/3000 and return: the bill won't be paid in this case because it falls outside the effective period of the recurring payment.

  If the proposed next pay date is after *end_date*, set the status of recurring payment to inactive and set the *next_pay_date* to 01/01/3000 and return.

  Otherwise, set the status of the recurring payment to "active" and set its *next_pay_date* to the proposed next pay date.

Recurring Payment Attached Accounts

It is possible to update the attached accounts after setting up a recurring payment. But in this case recurring payment can behave differently in different conditions, such as:

- If the recurring payment has already synchronized the bills for the accounts, but hasn't scheduled a payment, then the new changes will take effect from the next billing cycle.
- If the recurring payment hasn't synchronized the bills, or a payment has been made for the synchronized bills for the previous accounts, then the new changes will take effect from the next billing cycle.

The only difference in the above two cases is that at the first instance a payment is made for the bills from the old accounts even after the user has updated the accounts. So this can be confusing to the user. To overcome this situation, if the user changes the attached accounts an appropriate message can be displayed to the user. The getBillScheduled() method in the recurring payment object can be used to determine whether a payment is scheduled or not.

The main difference between a recurring payment for B2C and B2B is the availability of multiple billing accounts in B2B case. Inherently, the recurring payment object has a payer account number field which is the billing account a recurring payment supposed to handle. In the B2B scenario this field does not take the same meaning as in B2C, rather it becomes a reserved field for the payment core. The new field called attached accounts becomes more significant in the B2B case. This field accepts a set of IRecurringAttachedAccount objects the recurring payment has to handle. In the payment core, if the attached accounts are available, the payer account number field is populated with an internally generated ID. The payee ID field of the recurring payment object represents the payee DDN of the recurring payment. Other fields in the recurring payment object carry their usual meaning. Refer to the IRecurringPayment API for details.

The IRecurringAttachedAccount has the fields to represent the payer accounts attached to the recurring payment, including the account number and the corresponding Indexer DDN. Other fields in the recurring attached account object carry their usual meaning.

There is no difference in the PayServer API to be used to setup the recurring payment in B2C and B2B other than the getRecurringAttachedAccounts method.

When a recurring payment with multiple accounts is setup, the multiple accounts are inserted to the RECURRING_ATTACHED_ACCOUNTS table.

Deleting a Recurring Payment

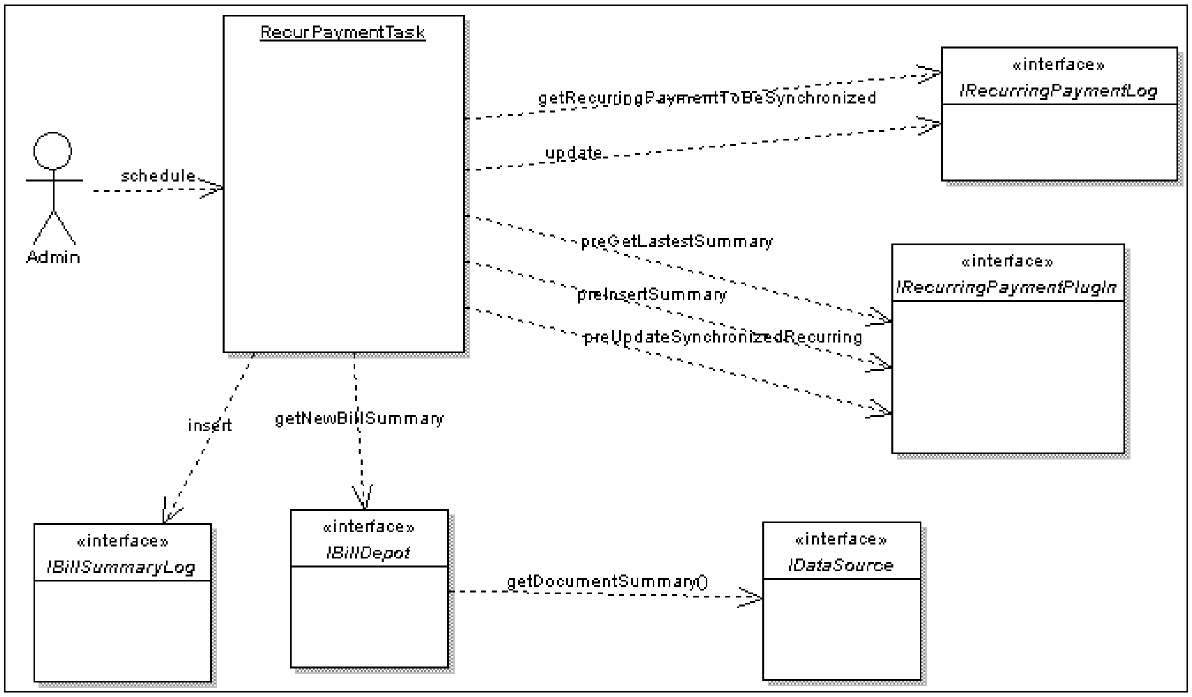
When a recurring payment is deleted, if the recurring payment has synchronized the bills but it still hasn't scheduled a payment, then the synchronized bill details are deleted from the PAYMENT_BILL_SUMMARIES & PAYMENT_ATTCHED_BILL_SUMMARIES tables.

Recurring Payment Back End Job

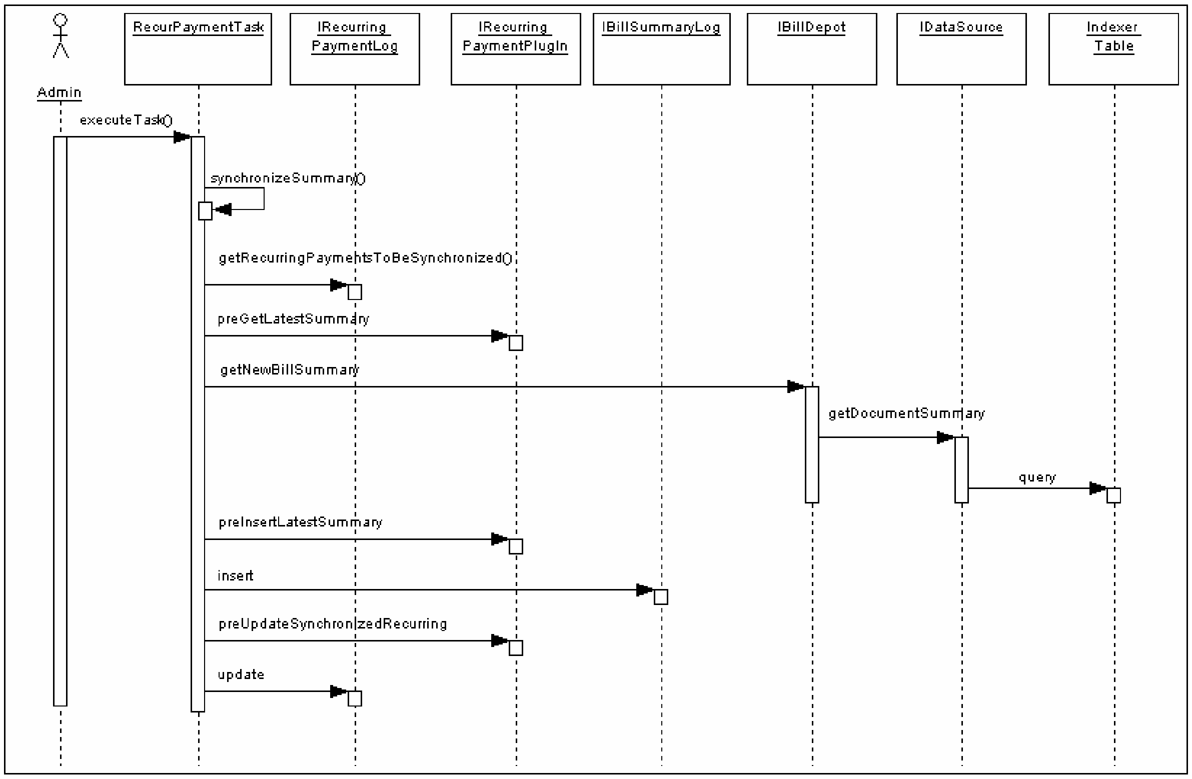
The pmtRecurringPayment job gets bills from eStatement Manager and then schedules payments. The first process is called "synchronization" and the second process is called "scheduling". In ePayment Manager version 4.2, these two processes are split into two separate tasks.

Recurring Payment Synchronization

During the synchronization process, the job retrieves a list of recurring payments to be synchronized, and then tries to get the bills for the recurring payments from eStatement Manager. The following UML diagram illustrates this process:



The following diagram shows the synchronization:



The following steps explain synchronization in B2C scenario:

1. RecurPaymentSynchronizerTask.executeTask() is called when the job runs, which calls RecurPaymentSynchronizerTask.synchronizeSummary().

2. RecurPaymentSynchronizerTask.synchronizeSummary() is called. This method does the real work of synchronization and following are the actions taken in this method.

3. IRecurringPaymentLog.getRecurringPaymentsToBeSynchronized() is called to get a list of recurring payments to be synchronized. The query result is affected by the recurring payment job configuration parameter "When to synchronize recurring payment with eStatement Manager". When this configuration is "whenever job runs", all the recurring payments are retrieved from the recurring_payments table with payee_id as the job DDN and status as "active". If "only after current bill is scheduled" is selected, then all the payments with the payee_id as job DDN and status as "active" and bill_scheduled as "Y" will be retrieved from the recurring_payments table.

4. For each recurring payment, IRecurringPaymentPlugIn.preGetLatestSummary() is called. This method allows the recurring payment plug-in code to decide whether to retrieve bills for a particular recurring payment based on biller-specific business rules.

5. Call RecurPaymentSynchronizerTask.updateRecuringPaymentOnly() if the plug-in rejects this recurring payment by returning PRE_GET_LATEST_SUMMARY_REJECT. This method does these things:

- Update *last_process_time* to the current time.
- If the recurring payment pay date is fixed date (monthly/quarterly/weekly) and pay amount is based on (minimum) amount due, and no bill arrives for this pay period (*bill_scheduled* is

"Y" and current time is after the *current next_pay_date*), the *last_pay_date* is updated to current *next_pay_date*. This ensures that if no bill arrives for this pay period; the next bill will be paid on the correct date.

- Call `IRecurringPayment.update()`: this method calculates the *next_pay_date* based on the *current last_pay_date*. See the preceding section for more information about how this update () operation works.

6. Call `IBillDepot.getNewBillSummary()`. This interface is implemented by *com.edocs.payment.imported.eadirect.BillDepot*. The `BillDepot` class retrieves the latest bill summary for the specified account.

- `BillDepot.getNewBillSummary()` is called, which then calls `BillDepot.getSummary()`
- BillDepot.getSummary() is called. This method calls IDataSource.getDocumentSummary() to get all the bills indexed for this account between the *last_process_time* of the recurring payment and the current job run time.
- The returned bills are in the format of name value pairs with value of string. They are interpreted to retrieve due date, amount due and/or minimum amount due.
  - **a** For each bill, if minimum amount due is not null, call `BillDepot.preParseMinAmountDue()` to give a child class of `BillDepot` (via the plug-in) a chance to manipulate the minimum amount due string before it is parsed, then it parses min amount due.
  - **b** If the bill's amount due is not null, call `BillDepot.preParseAmountDue()` to give child class of BillDepot (via the plug-in) a chance to manipulate the amount due string before it is parsed, then it parses the amount due. If the amount due fails to parse, the bill is ignored.
  - **c** If the bill has no amount due, or its amount due is set to null by `preParseAmountDue()`, or the amount due failed parsing, then the bill is ignored.
  - **d** If the bill's due date is not null, call `BillDepot.preParseDueDate()` to give child class of `BillDepot` (via the plug-in) a chance to manipulate the due date string before it is parsed, then it parses the due date.
  - **e** If the bill has no due date, or its due date is set to null by `preParseAmountDue()`, or the due date failed parsing, then the bill is ignored.
- All the successfully parsed bills are compared with the bill summary associated with the current recurring payment, if the summary is not null. The following business rules are used to decide which bill is the latest one:

  The due dates of the bill summaries retrieved are compared and the one with latest due date is chosen.

  For re-bill, multiple bills with the same due date may be retrieved. In this case, a re-bill is chosen based on the following rules: the one with latest doc date and in case of the same doc date, the one with the larger IVN number. This assumes that a re-bill is indexed after its original bill. A re-bill will be ignored if its original bill has been paid (the *bill_scheduled* flag of recurring payment is "Y").

- `BillDepot.Summary()` returns the latest bill if there is one found, otherwise, it returns null.

For B2B scenario, follow these steps to synchronize recurring payment:

1. `RecurPaymentSynchronizerTask.executeTask()` is called when the job runs, which calls RecurPaymentSynchronizerTask.synchronizeSummary().

2. `RecurPaymentSynchronizerTask.synchronizeSummary()` is called. This method does the real work of synchronization and following are the actions taken in this method.

3. `IRecurringPaymentLog.getRecurringPaymentsToBeSynchronized()` is called to get a list of recurring payments to be synchronized. The query result is affected by the recurring payment job configuration parameter "When to synchronize recurring payment with eStatement Manager". When this configuration is "whenever job runs", all the recurring payments are retrieved from the *recurring_payments* table with *payee_id* as the job DDN and status as "active". If "only after current bill is scheduled" is selected, then all the payments with the *payee_id* as job DDN and status as "active" and *bill_scheduled* as "Y" will be retrieved from the *recurring_payments* table.

4. For each recurring payment, get the set of attached accounts and iterate through attached accounts. For each attached account `IRecurringPaymentPlugIn.preGetLatestSummary()` is called. This method allows the recurring payment plug-in code to decide whether to retrieve bills for a particular recurring payment based on biller-specific business rules.

5. Call `RecurPaymentSynchronizerTask.updateRecuringPaymentOnly()` if the plug-in rejects this recurring payment by returning PRE_GET_LATEST_SUMMARY_REJECT. This method does these things:

- Update *last_process_time* to the current time.
- If the recurring payment pay date is fixed date (monthly/quarterly/weekly) and pay amount is based on (minimum) amount due, and no bill arrives for this pay period (*bill_scheduled* is "Y" and current time is after the *current next_pay_date*), the *last_pay_date* is updated to current *next_pay_date*. This ensures that if no bill arrives for this pay period; the next bill will be paid on the correct date.
- Call `IRecurringPayment.update()`: this method calculates the *next_pay_date* based on the *current last_pay_date*. See the preceding section for more information about how this update () operation works.

6. Call `IBillDepot.getNewBillSummary()` to get the bill summary for the attached account. This interface is implemented by *com.edocs.payment.imported.eadirect.BillDepot*. The `BillDepot` class retrieves the latest bill summary for the attached account.

- `BillDepot.getNewBillSummary()` is called, which then calls `BillDepot.getSummary()`
- BillDepot.getSummary() is called. This method calls IDataSource.getDocumentSummary() to get all the bills indexed for this account between the *last_process_time* of the recurring payment and the current job run time.
- The returned bills are in the format of name value pairs with value of string. They are interpreted to retrieve due date, amount due and/or minimum amount due.

  A. For each bill, if minimum amount due is not null, call `BillDepot.preParseMinAmountDue()` to give a child class of `BillDepot` (via the plug-in) a chance to manipulate the minimum amount due string before it is parsed, then it parses min amount due.

  B. If the bill's amount due is not null, call `BillDepot.preParseAmountDue()` to give child class of BillDepot (via the plug-in) a chance to manipulate the amount due string before it is parsed, then it parses the amount due. If the amount due fails to parse, the bill is ignored.

  C. If the bill has no amount due, or its amount due is set to null by `preParseAmountDue()`, or the amount due failed parsing, then the bill is ignored.

  D. If the bill's due date is not null, call `BillDepot.preParseDueDate()` to give child class of `BillDepot` (via the plug-in) a chance to manipulate the due date string before it is parsed, then it parses the due date.

  E. If the bill has no due date, or its due date is set to null by `preParseAmountDue()`, or the due date failed parsing, then the bill is ignored.
- All the successfully parsed bills are compared with the bill summary associated with the current recurring payment, if the summary is not null. The following business rules are used to decide which bill is the latest one:
- The due dates of the bill summaries retrieved are compared and the one with latest due date is chosen.
- For re-bill, multiple bills with the same due date may be retrieved. In this case, a re-bill is chosen based on the following rules: the one with latest doc date and in case of the same doc date, the one with the larger IVN number. This assumes that a re-bill is indexed after its original bill. A re-bill will be ignored if its original bill has been paid (the *bill_scheduled* flag of recurring payment is "Y").
- `BillDepot.Summary()` returns the latest bill if there is one found, otherwise, it returns null.

The above steps 4, 5 and 6 are repeated for each attached account of the recurring payment. For each recurring payment, the attached bill summaries are stored to a list. A new bill summary object is created for consolidated bill summary of the recurring payment. The attached bill summaries list is assigned to the consolidated bill summary object by calling `BillSummary.setAttachedBillSummaries(List attachedBillSummaries)` method.

The following steps are executed for both B2C and B2B scenarios:

1. Call `RecurPaymentTask.isValidBillSummary()` to validate the latest retrieved bill summary.

The latest bill summary could be ignored if it has no bill due date, or if the recurring payment is based on minimum amount due but the bill summary has no minimum amount due, or the recurring payment is based on amount due but the bill summary has no amount due.

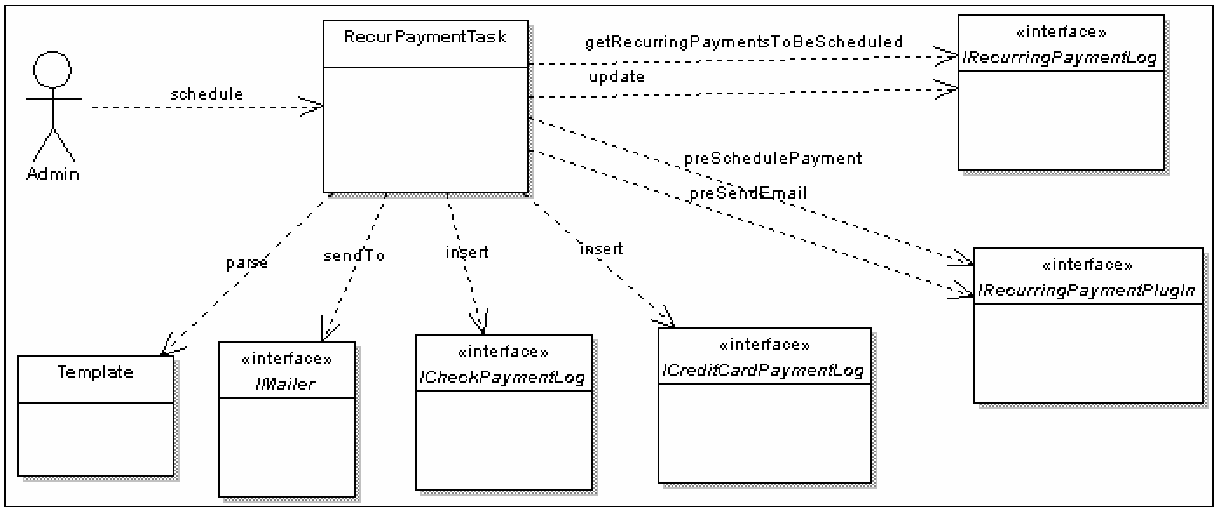
2. Now we have a valid bill summary. If the payment to the previous bill summary is still in "scheduled" status, do following:

- Call `RecurPaymentSynchronizerTask.cancelScheduledPayment()` to cancel this payment. The reason to cancel it is that the new bill summary just retrieved should include the balance of this scheduled bill, and we need to cancel the payment so that we won't pay the same bill twice.
- Call `RecurPaymentSynchronizerTask.modifyLastPayDate()`: If a recurring payment has a fixed pay date, but the amount is based on amount due or minimum amount due, we need to back date the last pay date because the previous bill payment has been cancelled. Failing to do so will cause the current new bill being paid in next pay interval, not the current one. For example, assume that current bill cycle is October, the previous bill was retrieved on Oct 10 and is scheduled to pay on Oct 15. As a result, the *last_pay_date* and *next_pay_date* of the recurring payment are updated to Oct 15 and Nov 15, respectively. On Oct 11, a new bill is retrieved and the payment is scheduled. If we don't back up the *last_pay_date*, the new bill will be scheduled to pay on Nov 15. But in this case, we do want to pay the bill on Oct 15 because we are still in the Oct billing cycle. To fulfill this goal, we are going to back date the *last_pay_date* to Sep 15 so the *next_pay_dat*e will be calculated as Oct 15, which will be used as the pay date for the new bill.
- Call `RecurPaymentSynchronizerTask.insertNewBillAndUpdateRecurring()`, which inserts the retrieved new bill and updates recurring payment accordingly.
- Call `IRecurringPaymentPlugIn.preInsertLatestSummary()` before inserting the bill summary in the *payment_bill_summaries* table.
- If PRE_INSERT_LATEST_SUMMARY_REJECT is returned from the plug-in, call `RecurPaymentSynchronizerTask.updateRecurringPaymentOnly()` and return. See step 5 for details about what this method does.
- Call `IBillSummaryLog.insert()` to insert this new bill summary.
- If `IBillSummaryLog.insert()` throws DuplicateKeyException indicating that this bill is already in the database, so call `RecurPaymentSynchronizerTask.updateRecurringPaymentOnly()`. See step 5 for details about what this method does.
- Set the *bill_scheduled* flag to "N" if the payment amount is not negative, or "Y" if it is negative. This means that no credit/reversal will be issued from recurring payment; the credit should show up as part of the next bill.
- Set the *bill_id* of the recurring payment to the one of the new bill summary.
- Call `IRecurringPaymentPlugIn.preUpdateSynchronizedRecurring()`.
- If PRE_UPDATE_SYNCHRONIZED_RECURRING_REJECT is returned from the plug-in, call RecurPaymenTask.updateRecurringPaymentOnly() and return. See step 5 for details about what this method does.
- Call `IRecurringPaymentLog.update()` to update the recurring payment. The following table lists the information being updated:

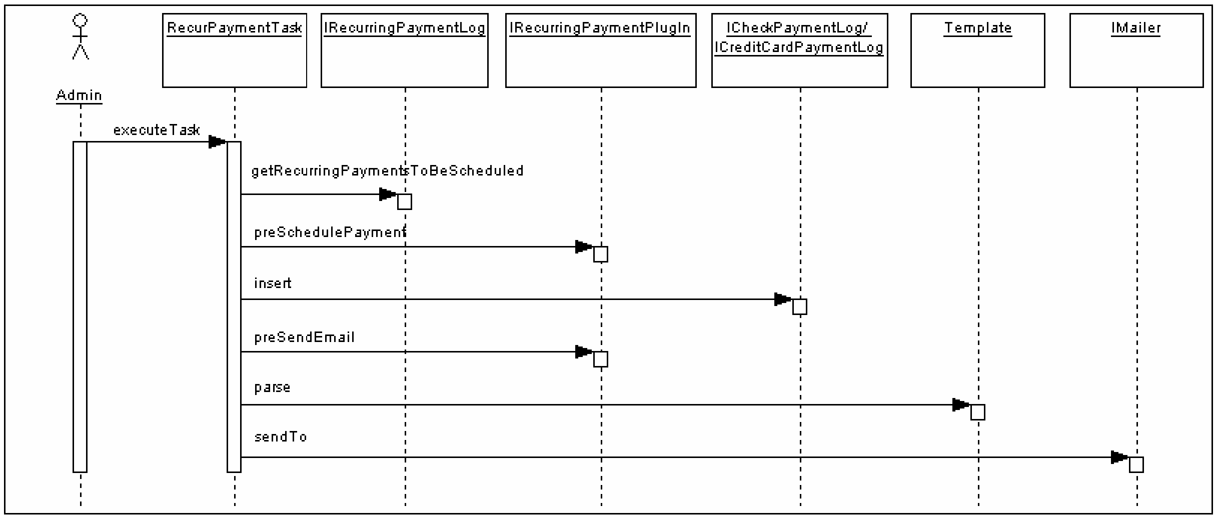
Column	**Value**
last_pay_date	In the case where the pay date is fixed, but amount is based on amount due, *last_pay_date* could be moved one *pay_interval* back if a scheduled payment is cancelled because a new bill arrives. Otherwise, *last_pay_date* will stay the same.
next_pay_date	*Next_pay_date* will be updated in `RecurringPaymentUtil.calculateInternal()`. In the case of fixed pay date, it will be updated based on *last_pay_date*; in case of before due, it will be updated based on the due date of the new bill. See the previous section, *Update Recurring Payment from the UI* on page 45 for more information.
status	Since *next_pay_date* is changed, the status could be changed to “inactive” if *next_pay_date* falls after *end_date*.
bill_id	It is set to the bill_id (doc ID) of the bill being inserted into the *payment_bill_summaries* table.
bill_scheduled	The *bill_scheduled* flag is set to "N" if the payment amount is not negative, "Y" if it is negative.
last_process_time	Set to the current time.

Recurring Payment Scheduling

During scheduling processing, the recurring payment job retrieves a list of recurring payments to be scheduled, and then schedules them. The following UML diagram shows the objects involved in this process.



The following diagram shows the action sequence:



The following steps describe the details of the actions that occur during recurring payment scheduling process:

1. `RecurPaymentSchedulerTask.execute()` is called when the job starts.

2. `RecurPaymentSchedulerTask.schedulePayments()` is called to do the actual scheduling work.

3. `IRecurringPaymentLog.getRecurringPaymentsToBeScheduled()` is called to get a list of recurring payments to be scheduled. The result is affected by the recurring payment job configuration parameter "Number of days before pay date to schedule the payment", which is a number, N. The SQK query finds all the recurring payments where the payee_id is the job's DDN reference, *bill_scheduled* is "N" and *next_pay_date* is <= today + N.

4. `IPayUserAccountAccessor.getPaymentAccount()` is called to get the current payment account information associated with this recurring payment. A sanity check is done on the retrieved payment account and different actions can be taken based on the result:

- If no payment account has been retrieved, which means it has been deleted from database, then the current recurring payment setup will be de-activated (`IRecurringPaymentLog.update()` is called to update status to inactive) and no payment is scheduled.
- If the payment account is a check account, it's status is cancelled, and the job configuration parameter "Cancel recurring payment if payment account is canceled?" is true, then the current recurring payment setup is de- activated (IRecurringPaymentLog.update() is called to update status to inactive) and no payment is scheduled.
- If the payment account is a credit card account, it has expired, and the job configuration parameter "Cancel recurring payment if payment account is canceled?" is true, then the current recurring payment setup is de-activated (`IRecurringPaymentLog.update()` is called to update status to inactive) and no payment is scheduled.

5. `RecurPaymentSchedulerTask.createPaymentTransaction()` is called to create a new payment transaction (either a check or a credit card) with status as scheduled and pay date and amount as specified by recurring payment setup.

6. `IRecurringPaymentPlugin.preSchedulePayment()` is called, which provides a way to customize the payment transaction before it is inserted into the database. If this method returns PRE_SCHEDUE_PAYMENT_REJECT, the payment won't be scheduled and the program returns to process next recurring payment; if not, the program goes to next step to schedule the payment.

7. Call `ICheckPaymentLog.insert()` to insert a check or `ICreditCardPaymentLog.insert()` to insert a credit card if the amount of the payment is not negative (Actually it will never be negative because the *bill_scheduled* won't be "N" if amount is negative. See job synchronization part for detail). Following table lists part of the payment information inserted into the payment tables:

Column	**Value**
status	6
pay_date	Should be the *next_pay_date* (calculated during synchronization process) of the current recurring payment. Since recurring payment will be updated after this insert operation, this value should actually be the same value as *last_pay_date* of the updated recurring payment.
Amount	This value is decided by *amount_type* and the amount of the recurring payment. It is calculated when RecurPaymentTask.createPaymentTransaction() is called. It should be the same as the amount column of the recurring payment if *amount_type* is “fixed”. It should be the same as the *amount_due* or *min_amount_due* of the bill associated with current recurring payment if *amount_type* is “amount due” or “minimal due”, respectively. If *amount_type* is “less due”, the payment amount is the amount due of the bill if amount due is less than or equal to the amount column value of the recurring payment. Otherwise, the payment amount value is 0. If *amount_type* is “upto amount”, then the payment amount is the amount due of the bill if amount due is less than or equal to the amount column value of the recurring payment. Otherwise, the payment amount is the amount column value of the recurring payment.
bill_id	Same as the one from recurring payment
Pid	Same as the one from recurring_payment
payer_id	Same as the one from recurring_payment
payer_acct_number	Same as the one from recurring_payment

8. IRecurringPaymentLog.update() is called to update the recurring payment. The following information of the recurring payment will be updated:

Column	**Value**
Curr_num_payments	Increased by 1.
Bill_scheduled	“N” if pay date is on fixed date (monthly, quarterly or weekly) and pay amount is fixed amount; “Y” otherwise.
Last_pay_date	The *last_pay_date* is set to the current *next_pay_date* of the recurring payment.
Next_pay_date	After *last_pay_date* is set to the current *next_pay_date*, the *next_pay_date* is calculated again by RecurringPaymentUtil.calculateInternal(). If the payment is using a fixed pay date (weekly, quarterly or weekly), then *next_pay_date* is calculated and moved to the next pay date in the next pay interval. In case of before due date, the next pay date will be calculated based on the current due date (whose bill has been paid), so this *next_pay_date* has no meaning until the next bill is synchronized.

Column	**Value**
Status	Status is re-calculated and will be changed to “inactive” if *next_pay_date* is after *end_date*, or *curr_num_payments* is greater than *max_num_payments*. See the previous section about UI update for details.

9. `IRecurringPaymentPlugIn.preSendEmail()` is called so that the plug-in can customize the email being sent out. The email won’t be sent out if this method returns PRE_SEND_EMAIL_REJECT.

10. `Template.parse()` is called to parse the email template and generate the content of email.

11. An email notification is sent to the user.

Important points about recurring payment scheduling:

- Payments are scheduled to the corresponding payment DDN (payee).
- The recurring payment object provides a convenient method *isAttchedAcountsAvailable()* to determine the nature of the recurring payment in both B2B and B2C. If the recurring payment is B2B, then *isAttchedAcountsAvailable()* method returns true and getAttachedAccounts method can be used to get the set RecurringAttachedAccount instances. Each of these RecurringAttachedAccount instances contains information such as the account number and the corresponding Indexer DDN. In B2C, *isAttchedAcountsAvailable()* method returns false and getPayerAcctNum() method can be used to get the account number. In B2C, the payee DDN is used as the Indexer DDN as well.
- When billing accounts from different billing cycles (bills are uploaded in different cycles) are added to a recurring payment, behavior of the recurring payment can be unpredictable depending on the recurring payment rules and the billing cycles. Common observations include:
  - The final schedule payment includes only a subset of accounts attached to the recurring payment.
  - Payments are scheduled late.
  - Some bills are skipped.

  It is highly recommended that accounts attached to a recurring payment belong to the same billing cycle. If it is needed to handle accounts from different billing cycles, then it should be tested properly to find the matching parameter combinations
- In the recurring payment scheduler task, when a payment is scheduled for multiple bills, individual bill details are inserted to the payment invoices table as invoices. The *AmountToBePaid* field of the Invoice object is populated for each type of recurring payment:
  - *IRecurringPayment.**AMOUNT_DUE***

    AmountToBePaid = Bill Amount Due
  - *IRecurringPayment.**MINIMAL_DUE***

    AmountToBePaid = Bill Min Amount Due

- *IRecurringPayment.**LESS_DUE***

  If the payment transaction amount is "0"

  AmountToBePaid = "0"

  Else

  AmountToBePaid = Bill Amount Due

- *IRecurringPayment.**UPTO_AMOUNT***

  In this case, the *AmountToBePaid* field is set to the bill amount due for each bill up to the amount specified in the recurring payment, and zero amount is set to the invoices corresponding to the remaining bills.

  For example, if there are four invoices at $40, $50, $30, and $20, and the up-to amount is $100, then it is distributed among the invoices as follows: $40, $50, $10, and $0. Note the last two invoices do not pay the total amount of the bill.

- *IRecurringPayment.FIXED_AMOUNT*

  This means to pay a fixed amount for the account regardless of the bill amount. If the payment is to be made before the due date of the bill, then invoices inserted would be only for the synchronized bills, else invoices would be inserted for each attached account. In either case the fixed amount would be equally distributed among the invoices and the last invoice amount would be adjusted for round-off errors.

Recurring Payment FAQ

This section answers a few common questions about recurring payment.

- Why is my current bill not paid by recurring payment after I set up my recurring payment?

  The recurring payment start date can only start from tomorrow, so the *last_process_date* is set to start from tomorrow. This means all the bills indexed before today won't be processed by the recurring payment. The reason is that, currently, there is no reliable way for recurring payment to know whether the current bill has been paid or not. The user may have paid it through a one time payment or through paper check. To avoid paying the bill twice, recurring payment will only start processing bills indexed since tomorrow.

  When a recurring payment is created, the JSP page checks whether there are any indexed bills for the account. If so, ePayment Manager retrieves the latest bill for the account. ePayment Manager also checks whether the latest bill has been paid by checking its doc ID against the *bill_id* of payment tables. If there is no match, we can reasonably assume that the bill has not been paid, so we prompt the user to make a one-time payment to pay that bill.

- What assumptions does recurring payment make about the bill system?

  Recurring payment assumes that the bill balances are accumulative; that is, the bill of this billing cycle includes the balance of the bill from previous billing cycle, and the later bill has a due date after that of the previous bill (the only case the same due date can happen is for re-bill, see below).

Recurring payment also assumes that each bill has a date indicating the chronological order of bills; this is usually the date when the bill arrives in the billing system. For example, in the case of eStatement Manager, doc date can be used to indicate the chronological order of bills arriving eStatement Manager. In the case of non-eStatement Manager billing system, other dates like statement date can be used for this purpose. When recurring payment synchronizes with eStatement Manager or other billing systems, it needs to retrieve the latest bill issued between the *last_process_time* and current time. This chronological date of bills (doc date or statement date) should be used to guarantee that functionality.

- Can recurring payment work with a billing system other than eStatement Manager?

  Yes. Recurring payment assumes nothing specific to eStatement Manager and the only thing you need to do is to re-implement the `IBillDepot` API. Of course, the billing system should meet assumptions stated in item 2.

- Do the bills need to have due dates?

  Yes, if the recurring payment is not fixed date and fixed amount. The due date is used to decide which bill is the latest one to pay. For eStatement Manager, you must index the due date or some date equivalent to use as the due date.

- What is rebill? How do I enable it?

  Re-bill means the same bill can be issued multiple times during one billing cycle to handle adjustments. All the re-bills must have the same due dates. To decide which re-bill is the latest bill to pay, the current `IBillDepot` implementation considers the one the latest with latest doc date. If there is more than one bill with same doc date, the bill with highest IVN number is chosen. Note: This implementation assumes that a later re-bill is always indexed after a previous re-bill, and no re-bills will be put together in one data file (which cause them have same doc date and IVN number). If you want to consider other factor such as amount for making the decision, you must re-implement `IBillDepot.`

  Re-bill is enabled by job configuration parameter "When to synchronize with eStatement Manager?" To use re-bill, you must choose "Whenever the job runs". If you don't have re-bill, you can choose either "whenever the job runs" or "only after current bill is scheduled".

  Technically, there is not much difference between a regular bill and re-bill. The major difference is the logic required to decide which re-bill is the latest bill, which goes beyond checking bill due date. You can think about non re-bill as a special case of re-bill: re-bill allows the same bill to appear more than once in a single billing period, but non re-bill appears only once. The code and programming logic actually doesn't distinguish between these two cases.

- When re-bill is not involved, is there any difference between the job configuration options for the job configuration parameter" when to synchronize with eStatement?"

  It should not affect functionality, and you can choose either of them. But you should consider these two things:

  First, performance may be deteriorated by choosing "whenever the job runs" because instead of waiting until current bill is scheduled, the job will try to synchronize with eStatement for each recurring payment. This can be especially true if you are talking with a billing system other than eStatement Manager that may have a slow connection.

  Second, a scheduled payment may be cancelled because of an "unexpected" early-arrival of next bill. Because we only want to pay the latest bill, the scheduled payment will be cancelled and the new bill will be scheduled.

- Why and when can a scheduled payment be cancelled by recurring payment job?

  The cancellation of a scheduled payment can only happen when the job configuration, "when to synchronize with eStatement" is set to "whenever job runs".

  It can happen because of two reasons:

  The first case is: (for re-bill) after the original bill is scheduled, but before it is processed, the re-bill arrives. In this case, the original payment will be cancelled, and the re-bill will be scheduled.

  Second, the bill of this billing cycle is still scheduled, but before it is processed, the bill of next billing cycle arrives (early). In this case, this bill's payment is cancelled and the next bill is scheduled.

  In case of fixed pay date and pay amount due, if a scheduled payment is cancelled, the *last_pay_date* and *next_pay_date* should all be move back by the *pay_interval* before the next bill is scheduled. This ensures that the next bill is paid with the same pay date as the previous bill.

- In the case of fixed pay date and pay amount due, what happens if there is no bill for this billing cycle?

  Recurring payment can never be triggered for a billing cycle if there is no bill, or if the bill's balance is negative (recurring payment doesn't issue credit). For example, a user sets to pay the bill's amount due on the 15th of each month, and current month is Oct. The *next_pay_date* will be set to Oct 15. However, if no bill arrives before Oct 15, then after Oct 15, the *next_pay_date* will be changed to Nov 15 to ensure that the bill arrives it will be paid in the next pay period. Otherwise, the user may end up paying the Nov bill with Oct pay date.

- Will recurring payment make a pay if the balance is negative?

  No. Instead, recurring payment assumes that this credit will roll into the balance of next bill. However, a zero dollar payment will be made if the balance is zero.

- Can I set up a recurring payment to pay from multiple payment accounts?

  Yes, in B2B scenario, you can set up a recurring payment for multiple billing accounts. The biller accounts (attached accounts) will be stored in *recurring_attached_accounts* table.

- Why does the default recurring payment update UI limit some options after the recurring payment is started? For example, it is not possible to switch from "pay on fixed date" to "pay before due".

  The logic to calculate next pay date becomes extremely complicated, so it is disallowed. If a custom UI does allow such update, the behavior is undefined.

- What happens if my credit card account expires?

  The recurring payment won't schedule a payment. It is then be de-activated and an email is sent to the user to indicate that he/she needs to update their credit card account info. In this case, the user must log on to cancel the inactive recurring payment and create a new one.

- Why wasn't my bill scheduled?

  This is the most often asked question, but there can be many causes. So here are offer a few hints to debug this problem. To start, review the recurring payment logic steps described previously.

First, check whether this is a false alarm. A bill can be synchronized, but yet scheduled. Also check the *next_pay_date* to see whether it reflects the correct pay date for the bill.

If the bill is not even synchronized, check whether it has been indexed;

If indexed, check whether it falls into the synchronization period. Only bills whose doc date fall between *last_process_time* and the current time will be considered.

Check whether this bill has valid information, for example, whether its due date, amount due are valid parse-able strings. A bill with invalid bill info or a negative balance won't be paid.

Even though this is a valid bill, it may not still be paid because its due date is before the due date of the current bill associated with the recurring payment.

Custom plug-ins may be a factor. The custom code may not have been thoroughly tested, so check the plug-in the code carefully, especially if the custom plug-in is manipulating the bill's due date, amount due, or recurring payment information directly.

The bill may not be scheduled because the payment account has been cancelled or deleted or de-activated.

- Will a single recurring payment failure fail the whole recurring payment job?

  It should not, otherwise it's a bug. If this happens, contact Oracle Technical Support.

- What is bill ID?

  It's a unique ID used to identify each bill. In eStatement Manager, it is the doc ID.

- What is last process time? What is it used for?

  It is the time when the last recurring payment job ran. It is used to ensure that a bill is only retrieved once from eStatement Manager. ePayment Manager only retrieves bills indexed between the last process time and the current time. That is, bills whose doc date >= last process time and <= current time. Previous versions of ePayment Manager also had time information as part of the last process time, but as of ePayment Manager 4.0, the last process time only contains date information (because the doc date only contains date information).

- What happens if a bill is indexed twice?

  This is similar to re-bill. The two bills have the same due dates, but the second indexing produces a later doc date, or a larger IVN, if they are indexed in the same day.

  If "when to synchronize with eStatement" is set to "whenever job runs", this is a true re-bill case, and will be treated as a re-bill.

  If "when to synchronize with eStatement" is set to "after current bill is scheduled", the second indexed bill will be ignored during next round of synchronization.

5 Sample User Interface

Customizing the ePayment Manager Front-End

If you choose the multiple DDN enrollment model you can start customizing *war-payment-complex.war of ear-payment-complex.ear* file. *war-payment.war* of *ear-payment.ear* includes the ePayment Manager Command Center JSP pages, which you usually don't need to change.

Customizing *web.xml*

Each WAR file includes a *WEB-INF/web.xml* file, which customizes the behavior of the WAR file. The following table lists some of the initialization parameters that you may want to change in *web.xml*. The PaymentServlet is used as an example:

Name	**Value**	**Description**
ServletRoot	com.edocs.payment.app	Package name of the payment servlet.
UserJspPath	*/payment/user/jsp/enroll/user/single_DDN* for single DDN or */paymentuser//jsp/enroll/user/multiple_DDN* for multiple DDN	Specifies where to get user enrollment JSP pages.
user_account_accessor	*com.edocs.payment.payenroll.usracct JNDISingleDDNUserAccountAccessor* for single DDN *com.edocs.payment.payenroll.usracct.JNDIMultipleDDNUserAccountAccessor* for multiple DDN	Implementation class of *IUserAccountAccessor*. This interface defines how to access user account information such as account number and email addresses. This value is configurable in the Payment Settings.
payment_account_accessor	*com.edocs.payment.payenroll.payacct.SSOPaymentAccountAccessor*	Implementation class of *IPaymentAccountAccessor*, which defines how to access payment account information. This value is configurable in the Payment Settings.
ErrorPage	*/payment/user/jsp/error.jsp*	Error page used when an error occurs.

Name	**Value**	**Description**
LoginRoot	*com.edocs.app.enrollment*	Login root (where the Login class sit).
LoginPage	*/paymentuser//jsp/enroll/user/single_DDN/ UserLogin.jsp* for single-DDN */payment/user/jsp/enroll/user/multiple_DDN/ UserLogin.jsp* for multiple-DDN,	Login page used for a particular enrollment model.
Account.name	*edx/paymentComplex/ejb/CDAAccount*	`IAccount` bean name.
TimeZone		Set this value if the Web server of the JSP pages is sitting on a different time zone other than the app server.
com.edocs.payment. payenroll.cache	True	Determines whether to cache `IPayUserAccount` into session. Currently, this value must be true.

Customizing the ePayment Manager JSPs

The ePayment Manager JSPs can be customized. The following table lists the ePayment Manager JSPs for ePayment Manager, along with a simple description. Most of the JSPs are under *web/payment/user/jsp*.

JSP Name	**Description**
cancelPayment.jsp	Displays the check or credit card payment to be cancelled.
cancelPaymentResponse.jsp	Response page when a check or credit card is cancelled.
confirmInvoice.jsp	*viewInvoice.jsp* is submitted to this page, which confirms the invoices to be paid.
error.jsp	Error page when there is a Payment error.
issueCredit.jsp	Issues credit to check account or reversal to credit card account
instantPayment.jsp	Creates and submits an instant payment.
schedulePayment.jsp	Schedules a payment, for either check or credit card. Collects payment information, such as amount and pay date.
schedulePaymentResponse.jsp	Response page after submitting *schedulePayment.jsp*.

JSP Name	**Description**
paymentHistory.jsp	Displays a list of processed/paid/returned/failed/cancelled check/credit card payments.
paymentReminder.jsp	Payment reminder setup page; collects payment reminder information.
paymentReminderReponse.jsp	Response page when *paymentReminder.jsp* page is submitted.
paymentReminderSummary.jsp	Lists all the reminders for the current user.
recurringPaymentCreate.jsp	Create a new recurring payment.
recurringPaymentUpdateDelete.jsp	Update an existing recurring payment.
recurringPaymentResponse.jsp	Response page when creating or updating a recurring payment.
recurringPaymentSummary.jsp	Lists all the recurring payments for the current user.
viewInvoice.jsp	Provides a list of invoices for testing the invoice feature. To use invoices, you must generate a page equivalent to *viewInvoice.jsp* with all the http form information in it. Note: This page transfers invoice data as an http form field array to *confirmInvoice.jsp*.
viewInvoice2.jsp	A simplified version of *viewInvoice.jsp*, which displays the minimum, required information for an invoice.
unknowForm.jsp	Displayed when the Payment servlet cannot process the request.
futurePayment.jsp	Displays a list of scheduled payments.
viewAchReturnError.jsp	Views ACH return pages.
viewInvoiceForPayment.jsp	Displays the invoices associated with a payment.
Detail.jsp	Equivalent to *eStatement Detail.jsp* with a few minor changes. Displays bill detail.
HistoryList.jsp	Equivalent to *eStatement HistoryList.jsp* with a few minor changes. Displays bill summary.
cancelExternalPayment.jsp	Displays the external payment to be cancelled.
cancelExternalPaymentResponse.jsp	Displays the external payment to be cancelled.
externalPayeeForm.jsp	External payee setup page; collects external payee information.
externalPayeeResponse.jsp	Schedules an external payment, for either check or credit card. Collects external payment information.
externalPaymentForm.jsp	Displays a list of scheduled/processed/paid/returned/failed/cancelled check/credit card external payments.

JSP Name	Description
externalPaymentHistory.jsp	Displays a list of scheduled/processed/paid/returned/failed/cancelled check/credit card external payments.
externalPaymentResponse.jsp	Response page after submitting *externalPaymentForm.jsp*.
externalPaymentSummary.jsp	Displays a list of external payees and provides links to manage external payees and to make external payments.

All payment JSP pages can only be accessed after login.

Zero amount check

By default, the JavaScript in *payCheck.jsp* prevents a zero dollar check from being created. To allow a zero amount check, change the JavaScript in this page to allow zero amount checks. A zero amount check will not be actually submitted to the payment gateway. Zero amount checks are used as records, and their statuses are changed to "processed" when pmtCheckSubmit runs.

Customized checks

There are five fields in the *check_payments* table which are customizable: *txn_timestamp_1*, *txn_time_stamp_2*, *flexible_field_1*, *flexible_field_2*, and *flexible_field_3*. They can be set using *payCheck.jsp*. See *payCheck.jsp* for details.

6 ePayment Manager Plug-ins

Plug-In Customization

The ePayment Manager plug-in is a callback, which allows you to add code to extend the functionality of ePayment Manager. ePayment Manager can use the following plug-ins:

- **`IAchCheckSubmitPlugIn`** for the ACH cartridge when submitting checks to ACH.
- **`IVerisignCreditCardSubmitPlugIn`** for the VeriSign cartridge when submitting credit cards to VeriSign.
- **`IPaymentReminderPlugIn`** for the job pmtPaymentReminder
- **`IRecurringPaymentPlugIn`** for the job pmtRecurPayment
  - **`IRecurPaymentSyncPlugIn`** for the pmtRecurPayment job Synchronizer task
  - **`IRecurPaymentSchedulePlugIn`** for the pmtRecurPayment job Scheduler task

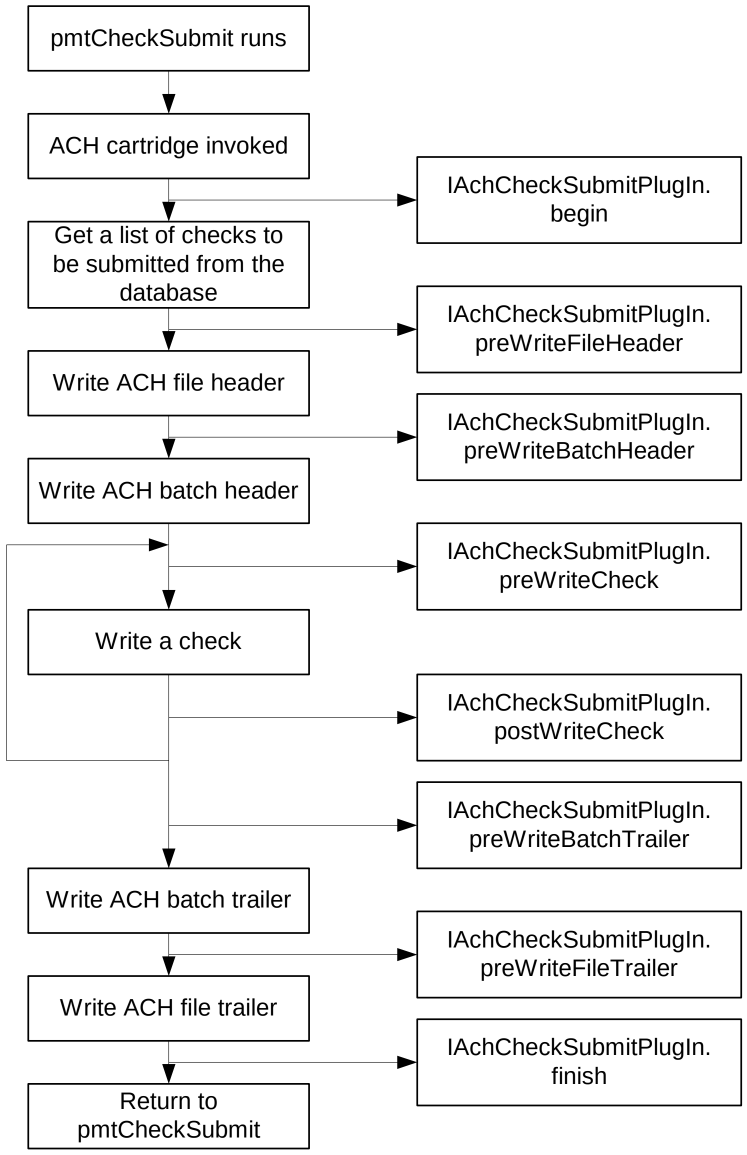
For each plug-in, ePayment Manager provides a default implementation. We recommend that you derive your plug-in from the default implementation to ensure that future updates to the plug-in will not break your code. The plug-ins and sample code are provided in Plug-in Sample Code.

ACH Check Submit Plug-in

Overview

The ACH cartridge supports a plug-in to modify ACH file generation. When the pmtCheckSubmit job runs for ACH, it calls the methods of the implementation of `IAchCheckSubmitPlugIn` (defined in Payment Settings) during numerous events. The default implementation is `AchCheckSubmitPlugIn`, which does nothing.

The following diagram shows the workflow for the pmtCheckSubmit job plug-in:


pmtCheckSubmit runs
ACH cartridge invoked
IAchCheckSubmitPlugIn.begin
Get a list of checks to be submitted from the database
IAchCheckSubmitPlugIn.preWriteFileHeader
Write ACH file header
IAchCheckSubmitPlugIn.preWriteBatchHeader
Write ACH batch header
IAchCheckSubmitPlugIn.preWriteCheck
Write a check
IAchCheckSubmitPlugIn.postWriteCheck
IAchCheckSubmitPlugIn.preWriteBatchTrailer
Write ACH batch trailer
IAchCheckSubmitPlugIn.preWriteFileTrailer
Write ACH file trailer
IAchCheckSubmitPlugIn.finish
Return to pmtCheckSubmit

Writing a Plug-in

You can use the pmtCheckSubmit plug-in to change the default name of the ACH file, create a remittance file in addition to the standard ACH file, deny a check or change the default information put into the ACH file. You need to create your own implementation to accomplish these tasks. Refer to the ePayment Manager SDK JavaDoc for information about writing an implementation of `IAchCheckSubmitPlugIn`. To create your own implementation:

1. Derive your implementation from the default implementation `AchCheckSubmitPlugIn`.

2. Overwrite the methods whose behavior you wish to change.

3. When compiling, include *payment_common.jar* and *payment_client.jar* into your java classpath.

4. Package this class into *payment_custom.jar* of each EAR file. See *Packaging payment Custom Code* on page 137 for information about redeploying EAR files.

5. Change the Payment Settings to point to your new class.

Using a Plug-in to Write ACH Addenda Records

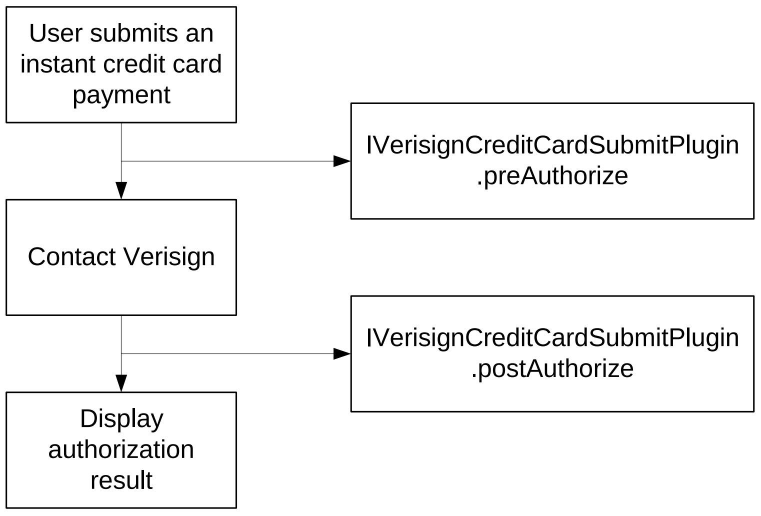
You can use the pmtCheckSubmit plug-in to write addenda records for ACH. The implementation called `AddendaCheckSubmitPlugIn` gets the invoice information of a payment and writes them out as addenda records. Check this class in the JavaDoc for its implementation details, and then follow the steps in *Writing a Plug-in* on page 70 to write your own implementation.

VeriSign Credit Card ePayment Manager Plug-in

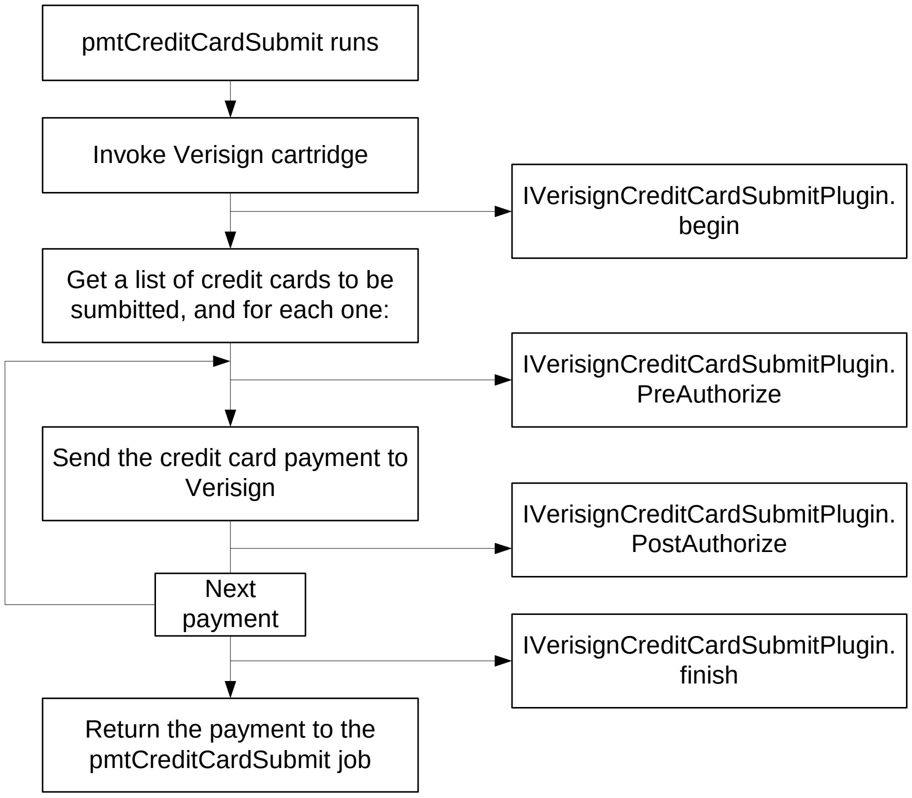
CreditCardSubmit Plug-in Overview

Unlike the ACH plug-in, the VeriSign credit card plug-in is invoked from both the front end (when an instant credit card is made) and the back end (when credit card submit job runs). This plug-in allows you to audit the credit card payment, deny it, or even changes the HTTP request sent to Verisign HTTP server. Check the API `IVerisignCreditCardSubmitPlugIn` for details.

The following diagram shows the workflow of the plug-in when an instant credit card payment is submitted:


User submits an instant credit card payment
IVerisignCreditCardSubmitPlugin .preAuthorize
Contact Verisign
IVerisignCreditCardSubmitPlugin .postAuthorize
Display authorization result

The following diagram shows the workflow of the plug-in when the pmtCreditCardSubmit job runs for VeriSign:


pmtCreditCardSubmit runs
Invoke Verisign cartridge
IVerisignCreditCardSubmitPlugin. begin
Get a list of credit cards to be sumbitted, and for each one:
IVerisignCreditCardSubmitPlugin. PreAuthorize
Send the credit card payment to Verisign
IVerisignCreditCardSubmitPlugin. PostAuthorize
Next payment
IVerisignCreditCardSubmitPlugin. finish
Return the payment to the pmtCreditCardSubmit job

Writing a Credit Card Plug-in

The default implementation of `IVerisignCreditCardSubmitPlugIn`, `VerisignCreditCardSubmitPlugIn`, just does nothing. To write you own implementation, you should:

1. Derive your implementation from `VerisignCreditCardSubmitPlugIn`.

2. Overwrite the methods for which you wish to change the default behavior.

3. When compiling, include *payment_common.jar* and *payment_client.jar* in your javac class path.

4. Package this class into *payment_custom.jar* of each ear file. For details about how to do that, see the *Deploying and Customizing J2EE Applications Guide for Oracle Siebel eStatement Manager*.

5. Change the Payment Settings of that DDN to use the new plug-in implementation.

ePayment Manager Reminder Plug-in

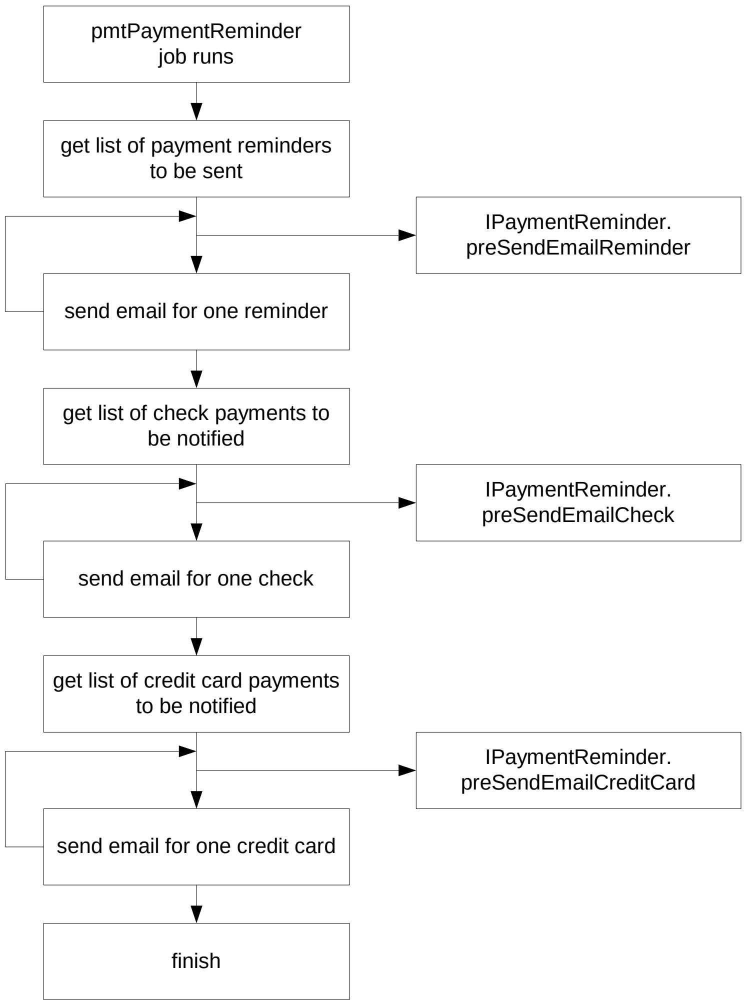
ePayment Manager Reminder Plug-in Overview

The ePayment Manager reminder plug-in is invoked when the pmtPaymentReminder job runs. pmtPaymentReminder does three things:

- Regular payment reminders
- Check status notification
- Credit card status notification

There are corresponding plug-ins for the preceding tasks. Refer to `com.edocs.payment.tasks.reminder.IPaymentReminderPlugIn` for details.

The following diagram shows the workflow for the plug-in of the pmtPaymentReminder job:



pmtPaymentReminder job runs
get list of payment reminders to be sent
IPaymentReminder. preSendEmailReminder
send email for one reminder
get list of check payments to be notified
IPaymentReminder. preSendEmailCheck
send email for one check
get list of credit card payments to be notified
IPaymentReminder. preSendEmailCreditCard
send email for one credit card
finish

Creating a pmtPaymentReminder Plug-in

The default plug-in implementation, `com.edocs.payment.tasks.reminder.PaymentReminderPlugIn`, actually does nothing. To implement your own plug-in:

1. Derive your implementation class from `PaymentReminderPlugIn`.

2. Overwrite the methods for you wish to change behavior.

3. When compiling, include *payment_common.jar* and *payment_client.jar* in your javac class path.

4. Package this class into payment_custom.jar of each ear file. See *Deploying and Customizing J2EE Applications Guide for Oracle Siebel eStatement Manager*.

5. Update the *pmtPaymentReminder* job configuration to use the new class.

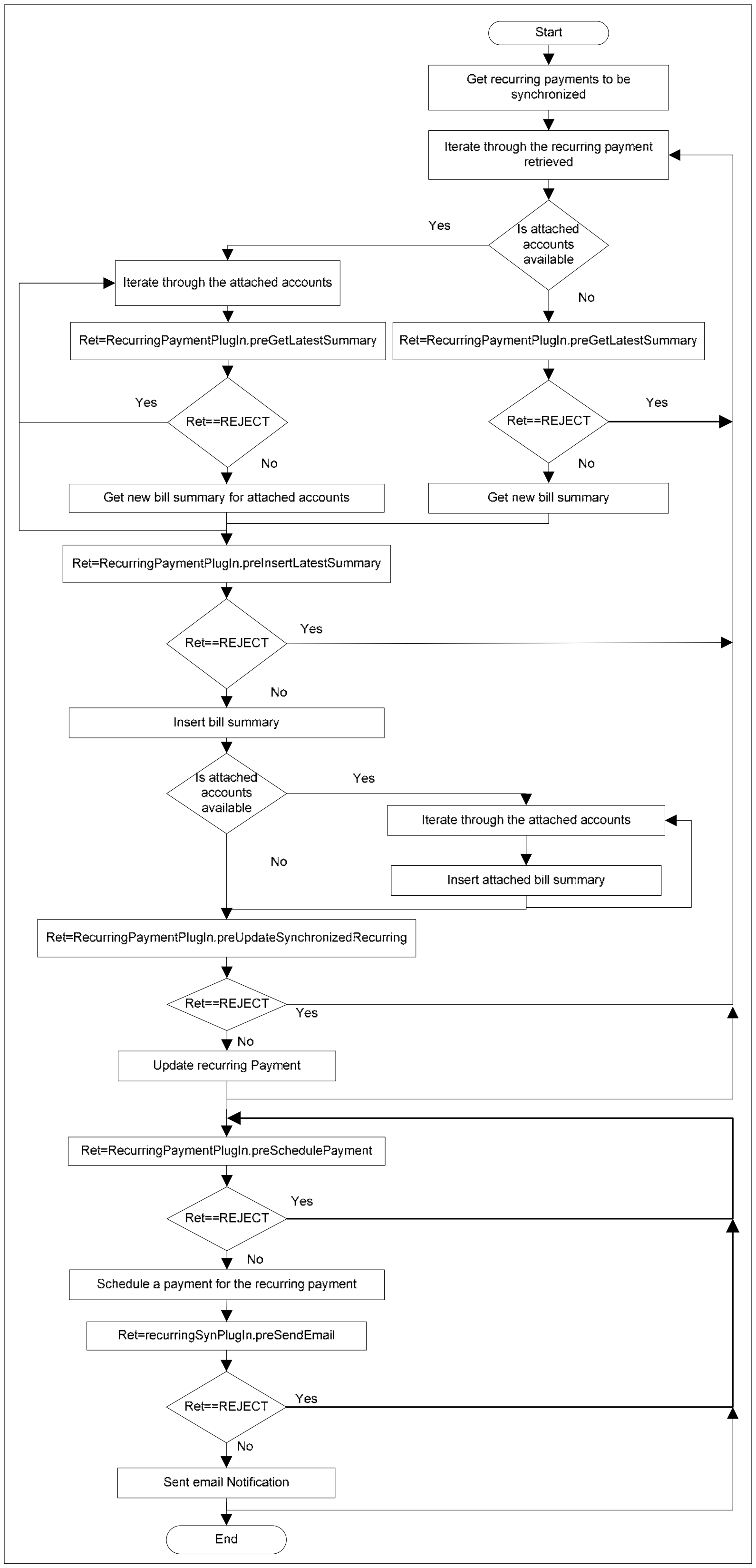
Recurring Payment Plug-in

Recurring Payment Overview

The recurring payment plug-in is called when the pmtRecurPayment job runs. You can use this plug-in to prevent a recurring payment from being scheduled based on business rules. Or, you can extract some indexed fields from the index table and put them into the payment being scheduled. The implementations: `com.edocs.tasks.payment.recur_payment.RecurringPaymentPlugIn`, is the default one and it does nothing.

The file *SampleRecurringPlugin.java* provides an example implementation.

The following diagram shows the workflow of recurring payment and how the plug-in works:


Start
Get recurring payments to be synchronized
Iterate through the recurring payment retrieved
Is attached accounts available
Yes
No
Iterate through the attached accounts
Ret=RecurringPaymentPlugIn.preGetLatestSummary
Ret=RecurringPaymentPlugIn.preGetLatestSummary
Ret==REJECT
Yes
No
Ret==REJECT
Yes
No
Get new bill summary for attached accounts
Get new bill summary
Ret=RecurringPaymentPlugIn.preInsertLatestSummary
Ret==REJECT
Yes
No
Insert bill summary
Is attached accounts available
Yes
No
Iterate through the attached accounts
Insert attached bill summary
Ret=RecurringPaymentPlugIn.preUpdateSynchronizedRecurring
Ret==REJECT
Yes
No
Update recurring Payment
Ret=RecurringPaymentPlugIn.preSchedulePayment
Ret==REJECT
Yes
No
Schedule a payment for the recurring payment
Ret=recurringSynPlugIn.preSendEmail
Ret==REJECT
Yes
No
Sent email Notification
End

Writing a Plug-in

The default plug-in implementation, `com.edocs.payment.tasks.recur_payment.RecurringPaymentPlugIn`, does nothing. To implement your own plug-in:

1. Derive your implementation class from `RecurringPaymentPlugIn`.

2. Overwrite the method that you want to change behavior of.

3. When compiling, include *payment_common.jar* and *payment_client.jar* in your javac class path.

4. Package this class into *payment_custom.jar* of each EAR file. For more information, see the *Deploying and Customizing J2EE Applications Guide for Oracle Siebel eStatement Manager*.

5. Update the pmtRecurPayment job configuration to use the new class.

Populating Index Fields into Payment Flexible Fields

`com.edocs.paymenttasks.recur_payment.SampleRecurringPlugIn` demonstrates how to use a plug-in to populate the flexible fields of the payment database (`ICheck` or `ICreditCard`) with the indexed information from the indexer table.

Recurring Payment Synchronizer Task Plug-in

The *IRecurPaymentSyncPlugIn* can be used to customize the synchronizer task. This plug-in interface provides the following methods:

```
public int preGetLatestSummary(SynchronizeRecurringPlugInParams p)
public int preInsertLatestSummary(SynchronizeRecurringPlugInParams p)
public int preUpdateSynchronizedRecurring(SynchronizeRecurringPlugInParams p)
```

The instance of `SynchronizeRecurringPlugInParams`, which is passed to these methods, contains information such as the RecurringPayment instance, the BillSummary instance, and the PaymentConfig instance as before. The following method was introduced to the SynchronizeRecurringPlugInParams to allow developers to get the indexer configuration for a given DDN:

```
public IPaymentConfig getIndexerConfig(String ddn)
```

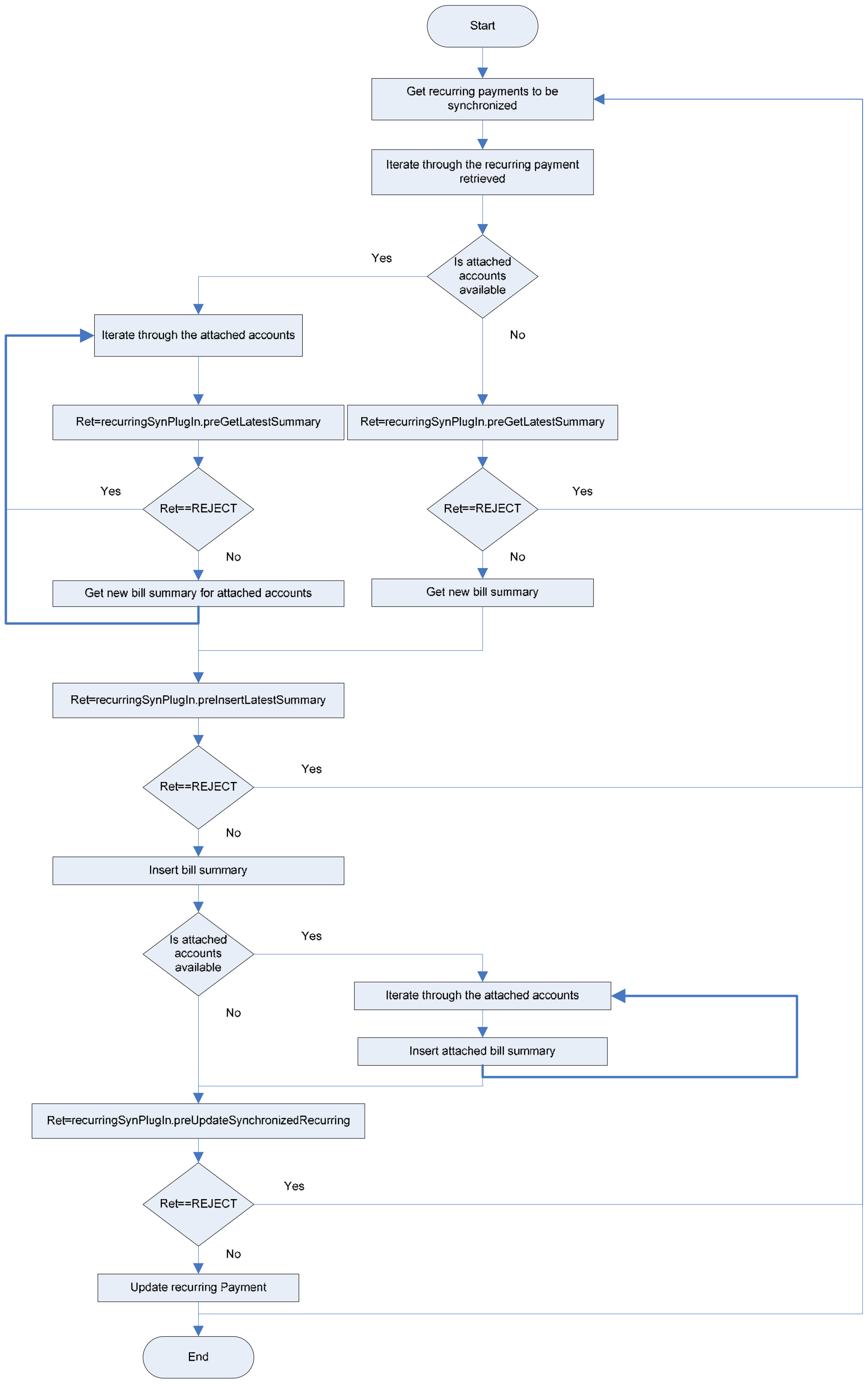
Note: This method retrieves Indexer configurations from a cache. If the Indexer configuration of the requested DDN is not cached yet, this method returns `null`.

When attached accounts are available (in B2B), preGetLatestSummary method is called before retrieving the bill summary of each attached account. Plug-in developers can use `p.getRecurringPayment().getCurrentAttachedAccount()` inside the preGetLatestSummary method to get the currently processing recurring attached account, when **p** is the `SynchronizeRecurringPlugInParams` instance passed to the preGetLatestSummary method. The recurring attached account instance contains information such as the account number and the Indexer DDN.

In a B2C system, `p.getRecurringPayment().getPayerAcctNum()` method can be used to get the account number. In this case, the Indexer DDN and payee DDN are always same.

In B2B, a bill is synchronized for each attached account (the account number and the indexer DDN of the attached account are used to retrieve the bill). In this case, as the bills can have different due dates, it is important to select a bill for evaluating the recurring payment rules, which depends on the bill due date. By default the payment module selects the bill with the minimum due date (earliest due date). This behavior can be modified in the `preInsertLatestSummary()` method. In this method one can get the list of bills synchronized and set the appropriate bill due date to the recurring payment object. This method can also be used to modify the default consolidated bill details set to the recurring payment object.

The following diagram shows the flow of the recurring payment synchronizer task:


Start
Get recurring payments to be synchronized
Iterate through the recurring payment retrieved
Is attached accounts available
Yes
No
Iterate through the attached accounts
Ret=recurringSynPlugIn.preGetLatestSummary
Ret=recurringSynPlugIn.preGetLatestSummary
Ret==REJECT
Yes
No
Ret==REJECT
Yes
No
Get new bill summary for attached accounts
Get new bill summary
Ret=recurringSynPlugIn.preInsertLatestSummary
Ret==REJECT
Yes
No
Insert bill summary
Is attached accounts available
Yes
No
Iterate through the attached accounts
Insert attached bill summary
Ret=recurringSynPlugIn.preUpdateSynchronizedRecurring
Ret==REJECT
Yes
No
Update recurring Payment
End

Recurring Payment Scheduler Task Plug-in

The role of the scheduler task is to schedule payments for the synchronized bills depending on the recurring payment rules (which amount to pay and when to pay). In a B2B system, this task uses the consolidated bill details as the basis for evaluating the recurring payment rules. When a payment is scheduled, the consolidate payment information is inserted to the appropriate payment table (CHECK_PAYMENTS or CREDITCARD_PAYMENTS). The individual bill details are inserted to the PAYMENT_INVOICES table as invoices attached to the payment. The `IRecurPaymentSchedulePlugIn` can be used to customize this behavior to some extent. This plug-in interface provides the following methods:

```
public int preSchedulePayment(SchedulePaymentPlugInParams params)
public int preSendEmail(SchedulePaymentPlugInParams params)
```

The task invokes the `preSchedulePayment` method of the plug-in before scheduling a payment and invokes the *preSendEmail* method before sending the notification. In the `preSchedulePayment()` method, the list of invoices can be accessed and additional details inserted into these invoices. Both of those methods receive an instance of `SchedulePaymentPlugInParams` as a parameter. Plug-in developers can use the following getter methods available on this parameter to access the corresponding domain objects:

```
public IPaymentConfig getPaymentConfig()
public IPaymentTransaction getPayment()
public IRecurringPayment getRecurringPayment()
public IPaymentAccount getPaymentAccount()|
public List getPaymentInvoices()
public HashMap getPropertyMap()
```

7 Customizing ePayment Manager Template Files

Overview

All the ePayment Manager email notifications are sent through IPaymentNotificationService. The default implementation of IPaymentNotificationService uses the XMA based notification module provided by eStatement Manager for sending email notifications. The eStatement Manager notification module uses Velocity (http://jakarta.apache.org/velocity/) as the template engine.

ePayment Manager provides another template engine to generate ePayment Manager-wide text messages, ACH files, and A/R files. This chapter describes how to customize both the Velocity based payment email templates and ePayment Manager templates provided by payment template engine.

Customizing Email Templates

ePayment Manager uses Velocity template files to generate customized text that will be sent in a notification email. The email templates can be customized for you by Oracle Professional Services, or you can customize them yourself. This appendix describes how email template variables and how they can be customized.

Please refer the Velocity User Guide (http://velocity.apache.org/) for more information on velocity templates. The Velocity template syntax is beyond the scope of this document.

Separate email notification templates are used for:

Type of notification	**Job that Specifies**	**Template File**
Reminder to pay bills and the status of the checks	pmtPaymentReminder	*payment_reminder_predue.xml.vm*
Fixed reminder setup by the user	pmtPaymentReminder	*payment_reminder_fixed.xml.vm*
Enrollment status	pmtNotifyEnroll	*payment_account_status.xml.vm*
Recurring payment was scheduled	pmtRecurPayment	*payment_recurring_scheduled.xml.vm*
Recurring payment was not scheduled	pmtRecurPayment	*payment_recurring_notscheduled.xml.vm*
Payment Command Center job status	All Payment jobs	*payment_job_notification.xml.vm*
Credit card expiration	pmtCreditCardExpNotify	*payment_ccaccount_ccexpired.xml.vm*

For UNIX, the default path to the email template files is *$EDX_HOME/template*
For Windows, it is:
%EDX_HOME%\template.

The Velocity template language includes a list of placeholders that refer to Java objects, which are hosted by Payment. It also includes some simple logic control directives such as *if* and *foreach*.

Payment Pre Due Reminder Template

Payment pre-due reminder messages are generated based on *payment_reminder_predue.xml.vm* which resides in $EDX_HOME/*template* (%EDX_HOME%\ *template* for Windows).

This template is used for regular payment reminder and email notifications for processed, returned or failed payments:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!-- edited with XMLSPY v5 rel. 4 U (http://www.xmlspy.com) by Usha (edocs) -
->

<message-config xmlns="http://www.edocs.com/messaging"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.edocs.com/messaging

content.xsd">

      <header>

            <subject>Payment Reminder</subject>

      </header>

      <content>

            <no>123</no>

            <type>html</type>

            <body><![CDATA[<html>

            #if($isCheck)

            Dear user $check.getPayerId():<br>

            #if($isPaid)

            <br>

            #if($isAmtNegative)

              A credit of $$check.getAmount().substring(1) has been issued to
your check account on
$messagingTemplateUtil.getFormattedDate($check.getPayDate(),"MMM dd yyyy").

            #else

              Your check of $$check.getAmount() has been paid on
$messagingTemplateUtil.getFormattedDate($check.getPayDate(),"MMM dd yyyy").

            #end

            <br>

            #end


            #if($isReturned)
            <br>

            #if($isAmtNegative)

              Your request to issue $$check.getAmount().substring(1) credit
to your check account has been rejected. The error message is:
$AchReturnCode.get($check.getTxnErrMsg()).

            #else

              Your check of $$check.getAmount() has been returned. The error
message is: $AchReturnCode.get($check.getTxnErrMsg())

            #end

            <br>

            #end


            #if($isFailed)

              <br>

              There is a problem to process your check. The error message is:
$check.getTxnErrMsg()

              <br>

            #end


            #if($isCanceled)

            <br>

            #if($isAmtNegative)

              Your request to issue $$check.getAmount().substring(1) credit
to your check account has been canceled by the payment system because the
check account is not valid. Please check your enrollment information.

            #else

              Your check of $$check.getAmount() has been canceled by the
payment system because the check account is not valid. Please check your
enrollment information.

            #end

            <br>

            #end


            #if($isProcessed)

            <br>
            #if($isAmtNegative)

              Your request to issue $$check.getAmount().substring(1) credit
to your check account has been sent to bank for clearing.

            #else

              Your check of $$check.getAmount() has been sent to bank for
clearing.

            #end

            <br>

            #end


            #end


            #if($isCCard)

            Dear user $creditcard.getPayerId():<br>

            #if($isSettled)

            <br>

            #if($isAmtNegative)

              Your request to reverse $$creditcard.getAmount().substring(1)
to your credit card has been authorized successfully.

            #else

              Your credit card payment of $$creditcard.getAmount() has been
authorized successfully.

            #end

            <br>

            #end


            #if($isFailed)

            <br>

            #if($isAmtNegative)

              Your request to reverse $$creditcard.getAmount().substring(1)
to your credit card failed authorization.<br>

              The error message is:  $creditcard.getTxnErrMsg()

            #else

              Your credit card payment of $$creditcard.getAmount() failed
authorization.<br>
```

```
              The error message is:  $creditcard.getTxnErrMsg()
            #end
            <br>
            #end


            #if($isSystemFailure)
            <br>
            #if($isAmtNegative)
              Your request to reverse $$creditcard.getAmount().substring(1)
to your credit card failed.<br>
              The error message is:  $creditcard.getTxnErrMsg()
            #else
              Your credit card payment of $$creditcard.getAmount()
failed.<br>
              The error message is:  $creditcard.getTxnErrMsg()
            #end
            <br>
            #end


            #if($isCanceled)
            <br>
            #if($isAmtNegative)
              Your request to reverse $$creditcard.getAmount().substring(1)
to your credit card has been canceled by the payment system because the
account is invalid. Please check your enrollment information.
            #else
              Your credit card payment $$creditcard.getAmount() has been
canceled by the payment system because the account is invalid. Please check
your enrollment information.
            #end
            <br>
            #end


            #end
            </html>
            ]]></body>
            <part>
                  <part_name>doc</part_name>
                  <part_type>html</part_type>
                  <part_body>Hello</part_body>
            </part>
            <part>
                  <part_name>pdf</part_name>
                  <part_type>html</part_type>
                  <part_body>Hello</part_body>
            </part>
      </content>
      <footer/>
</message-config>
```

The following table describes the payment pre due reminder template variables:

Variable	**Type**	**Description**
accountNumber	`String`	Biller account number
AchReturnCode	AchReturnCode	A new AchReturnCode Object
`check`	`ICheck`	The `ICheck` object being notified, valid only when `isCheck` is true.
`creditcard`	`ICreditCard`	The `ICreditCard` object being notified, valid only when isCCard is true.
`isCCard`	Boolean	True means this is for credit card status notification.
`isCheck`	Boolean	True means this is for check status notification.
`isFailed`	Boolean	True means the payment has failed to process.
`isPaid`	Boolean	True means the check has been paid or cleared.
`isProcessed`	Boolean	True means the check has been processed.
`isReminded`	Boolean	True means this is for regular payment reminders.
`isReturned`	Boolean	True means the check has been returned.
`isSettled`	Boolean	True means the credit card has been settled.
`isSystemFailure`	Boolean	True means there has been a system error. For example, a network failure.
payeeId	String	Payee ID

Variable	**Type**	**Description**
payerId	String	Payer ID
paymentId	String	Payment ID
`reminder`	`IPaymentReminder`	The `IPaymentReminder` object being reminded, valid only when `isReminded` is true.

Payment Fixed Reminder Template

Payment fixed reminder messages are generated based on *payment_reminder_fixed.xml.vm* which resides in $EDX_HOME/*template* (%EDX_HOME%\ *template* for Windows).

This template is used for regular payment reminder and email notifications for processed, returned or failed payments:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!-- edited with XMLSPY v5 rel. 4 U (http://www.xmlspy.com) by Usha (edocs) -
->

<message-config xmlns="http://www.edocs.com/messaging"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.edocs.com/messaging

content.xsd">

      <header>

            <subject>Payment Reminder</subject>

      </header>

      <content>

            <no>123</no>

            <type>html</type>

            <body><![CDATA[<html>

            #if($isRemind)

            Dear $reminder.getPayerId():<br>

            <br>

              This email is to remind you to pay your current
$reminder.getPayeeId()'s<br>

              bill. Please refer to this url to pay your bill:<br>

              http://www.oracle.com.<br>

              <br>

              Thanks,
            #end
            </html>
            ]]></body>
            <part>
                  <part_name>doc</part_name>
                  <part_type>html</part_type>
                  <part_body>Hello</part_body>
            </part>
            <part>
                  <part_name>pdf</part_name>
                  <part_type>html</part_type>
                  <part_body>Hello</part_body>
            </part>
      </content>
      <footer/>
</message-config>
```

The following table describes the payment fixed reminder template variables:

Variable	**Type**	**Description**
accountNumber	`String`	Biller account number
biller	`String`	Payee ID
billingURL	`String`	Billing URL
customerName	`String`	Payer ID
`isCCard`	Boolean	True means this is for credit card status notification.
`isCheck`	Boolean	True means this is for check status notification.
`isRemind`	Boolean	True means this is for regular payment reminders.
payeeId	`String`	Payee ID
payerId	`String`	Payer ID
`reminder`	`IPaymentReminder`	The `IPaymentReminder` object being reminded, valid only when `isReminded` is true.

Enrollment Notification Template

The enrollment notification template notifies customers about "active "and "bad-active" payment accounts and NOC returns. Enrollment reminder messages are generated based on *payment_account_status.xml.vm*:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!-- edited with XMLSPY v5 rel. 4 U (http://www.xmlspy.com) by Usha (edocs) -
->

<message-config xmlns="http://www.edocs.com/messaging"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.edocs.com/messaging

content.xsd">

      <header>

            <subject>Payment Account Notification</subject>

      </header>

      <content>

            <no>123</no>

            <type>html</type>

            <body><![CDATA[<html>

            Dear $checkAccount.getUserId():<br>

            <br>

            #if($isACH)

            #if($success)

              Your payment account $checkAccount.getAccountNumber() has been
succesfully activated.<br>

            #else

              There has been a problem activating your payment account
$checkAccount.getAccountNumber().<br>

              The return reason code is: $errCode<br>

            #end

            #end


            #if($isNOC)

            #if($isC01)

            <br>

            #if($isAutoUpdate)
```

```
  Your Bank Account Number has been changed.<br>
  New Bank Account Number is: $newPaymentAccount<br>
  Old Bank Account Number was: $oldPaymentAccount<br>
#else
  Your current Bank Account Number is out of date.<br>
  New Bank Account Number is: $newPaymentAccount<br>
  Current Bank Account Number is: $oldPaymentAccount<br>
  Please login to change your profile.<br>
#end
#end

#if($isC02)
<br>
#if($isAutoUpdate)
  Your Bank Routing Number has been changed.<br>
  New Bank Routing Number is: $newRouting<br>
  Old Bank Routing Number was: $oldRouting<br>
#else
  Your current Bank Routing Number is out of date.<br>
  New Bank Routing Number is: $newRouting<br>
  Current Bank Routing Number is: $oldRouting<br>
  Please login to change your profile.<br>
#end
#end

#if($isC03)
<br>
#if($isAutoUpdate)
  Your Bank Account Information has been changed.<br>
  New Bank Account Number is: $newPaymentAccount<br>
  Old Bank Account Number was: $oldPaymentAccount<br>
  New Bank Routing Number is: $newRouting<br>
  Old Bank Routing Number was: $oldRouting<br>
```

```
#else
  Your current Bank Account Information is out of date.<br>
  New Bank Account Number is: $newPaymentAccount<br>
  Current Bank Account Number was: $oldPaymentAccount<br>
  New Bank Routing Number is: $newRouting<br>
  Current Bank Routing Number is: $oldRouting<br>
  Please login to change your profile.<br>
#end
#end

#if($isC05)
<br>
#if($isAutoUpdate)
  Your Bank Account Information has been changed.<br>
  Your new Bank Type is $newPaymentType<br>
  Your old Bank Type was $oldPaymentType<br>
#else
  Your current Bank Account Type is out of date.<br>
  Your new Bank Type is $newPaymentType<br>
  Your current Bank Type is $oldPaymentType<br>
  Please login to change your profile.<br>
#end
#end

#if($isC06)
<br>
#if($isAutoUpdate)
  Your Bank Account Information has been changed.<br>
  New Bank Account Number is: $newPaymentAccount<br>
  Old Bank Account Number was: $oldPaymentAccount<br>
  Your new Bank Type is $newPaymentType<br>
  Your old Bank Type was $oldPaymentType<br>
#else
```

```
  Your current Bank Account Information is out of date.<br>
  New Bank Account Number is: $newPaymentAccount<br>
  current Bank Account Number is: $oldPaymentAccount<br>
  New Bank Type is $newPaymentType<br>
  Current Bank Type is $oldPaymentType<br>
  Please login to change your profile.<br>
#end
#end

#if($isC07)
<br>
#if($isAutoUpdate)
  Your Bank Account Information has been changed.<br>
  New Bank Account Number is: $newPaymentAccount<br>
  Old Bank Account Number was: $oldPaymentAccount<br>
  New Bank Routing Number is: $newRouting<br>
  Old Bank Routing Number was: $oldRouting<br>
  Your new Bank Type is $newPaymentType<br>
  Your old Bank Type was $oldPaymentType<br>
#else
  Your current Bank Account Information is out of date.<br>
  New Bank Account Number is: $newPaymentAccount<br>
  Current Bank Account Number is: $oldPaymentAccount<br>
  New Bank Routing Number is: $newRouting<br>
  Current Bank Routing Number is: $oldRouting<br>
  New Bank Type is $newPaymentType<br>
  Current Bank Type is $oldPaymentType<br>
  Please login to change your profile.<br>
#end
#end
#end

#if($isCDP)
```

```
            <br>

            #if($success)

              Your payment account $checkAccount.getAccountNumber() has been
succesfully activated.<br>

            #else

              There has been a problem activating your payment account
$checkAccount.getAccountNumber(). Please contact your customer service
representative for further assistance.<br>

            #end

            #end

            </html>

            ]]></body>

            <part>

                  <part_name>doc</part_name>

                  <part_type>html</part_type>

                  <part_body>Hello</part_body>

            </part>

            <part>

                  <part_name>pdf</part_name>

                  <part_type>html</part_type>

                  <part_body>Hello</part_body>

            </part>

      </content>

      <footer/>

</message-config>
```

This template is used for both ACH and Checkfree CDP. The text between `#if($isACH)` and the corresponding `#end` is for ACH. The text between `#if($isCDP)` and the corresponding `#end` is for Checkfree. If there are no payment gateways for Checkfree or for ACH, you can remove that section from the template file.

Each payment account will be sent an individual email. ePayment Manager supports multiple payment accounts, so there may be more than one email sent out for each customer (if that customer has multiple payment accounts).

The following tables list the variables available for use in the Enrollment Notification email template. The first table is for ACH, the second table is for ACH NOC returns, and the third table is for Checkfree CDP

The following variables apply to all the cases:

Variable	**Type**	**Description**
`checkAccount`	`ICheckAccount`	The current check account being notified
`template`	`Template`	The Payment template engine, which is used to declare new variables for the template.
`config`	`IPaymentConfig`	Payment setting information, which is configured from the Command Center.

The following variables apply to **ACH**:

ACH Variable	**Type**	**Description**
`isACH`	boolean	True indicates this is an ACH notification.
`success`	boolean	Success means this account has been activated successfully.
`errCode`	String	ACH return code, if the transaction failed.

The following variables apply to **ACH NOC** returns:

ACH NOC Variable	**Type**	**Description**
`isNOC`	boolean	True indicates this is an NOC return.
`isC01, isC02, isC03, isC05, isC06, isC07`	boolean	True indicates the returned NOC code(s).
`isAutoUpdate`	boolean	Returns the state of the *com.edocs.payment.cassette .ach.autoUpdatNOC flag*, which is configured on the Payment Settings page from the Command Center.
`newPaymentAccount`	String	New payment account number.
`oldPaymentAccount`	String	Old payment account number.
`newRouting`	String	New payment routing number.
`oldRouting`	String	Old payment routing number.
`newPaymentType`	String	New payment account type.
`oldPaymentType`	String	Old payment account type.

The following variables apply to **Checkfree CDP**:

CDP Variable	**Type**	**Description**
`isCDP`	Boolean	True indicate this is a Checkfree CDP notification.
`success`	Boolean	True indicates this account has been activated successfully.

Recurring Payment Notification Templates

When recurring payment schedules a payment, email notification messages are generated from the template file *payment_recurring_scheduled.xml.vm* whereas recurring payment not scheduled notification messages are generated from *payment_recurring_notscheduled.xml.vm* template file. The template file *payment_recurring_scheduled.xml.vm* is as follows.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!-- edited with XMLSPY v5 rel. 4 U (http://www.xmlspy.com) by Usha (edocs) -
->

<message-config xmlns="http://www.edocs.com/messaging"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.edocs.com/messaging

content.xsd">

      <header>

            <subject>Recurring Payment Scheduled</subject>

      </header>

      <content>

            <no>123</no>

            <type>html</type>

            <body><![CDATA[<html>

            Dear $recurringPayment.getPayerId(),<br>

            <br>

            #if($isPaymentScheduled)

              #if($isCheck)

                  This email is to inform you a check payment has been
scheduled automatically for you.<br>

            The check amount is $$payment.getAmount(). The pay date is
$messagingTemplateUtil.getFormattedDate($payment.getPayDate(),"MM/dd/yyyy").<
br>

              #end

              #if($isCCard)
```

```
                  This email is to inform you a credit card payment has been
scheduled automatically for you.<br>

            The credit card amount is $$payment.getAmount(). The pay date is
$messagingTemplateUtil.getFormattedDate($payment.getPayDate(),"MM/dd/yyyy").<
br>

              #end

                  You can update or cancel this transaction following this
URL http://www.oracle.com.<br>

            #end

            #if($isPaymentNotScheduled)

                  This email is to inform you that your recurring payment
scheduled on
$messagingTemplateUtil.getFormattedDate($recurringPayment.getNextPayDate(),"M
M/dd/yyyy")

            is not made as requested.<br>

             #if($isAmountNegative)

                  #set( $negativeAmount = $payment.getAmount() * -1 )

                  The bill showed an amount of -$$negativeAmount.<br>

             #end

                  Please contact your biller for details.<br>

            #end

            #if($isLessPayment)

                  This email is to inform you that the amount due
$$recurringPayment.getBillAmountDue() is

            more than the maximal amount, $$recurringPayment.getAmount(),
specified in the recurring payment.

            The bill is not paid by recurring payment.<br>

            #end

            #if($isAlreadyPaid)

                  The bill, due on
$messagingTemplateUtil.getFormattedDate($recurringPayment.getBillDueDate(),"M
M/dd/yyyy"), is not paid

            by recurring payment because it is already been paid.<br>

            #end

            #if($isLastRecurringPayment)

                  This is the last payment from the recurring payment.<br>

            #end
            #if($isRecurringPaymentCanceled)

                  This email is to inform you that your recurring payment
schedule has been deactivated,

            due to your account status, and no payments have been
scheduled.<br>

                  Please contact your biller for details.<br>

            #end

            #if($isUptoAmountExceeded)

                  Also, the amount due $$recurringPayment.getBillAmountDue()
is more than the maximal amount, $$recurringPayment.getAmount(),

            specified in the recurring payment. You still have an unpaid
balance of $$unpaidBalance.<br>

            #end

                  Number of payments made until now is
$recurringPayment.getCurrNumPayment().<br>

            </html>

            ]]></body>

            <part>

                  <part_name>doc</part_name>

                  <part_type>html</part_type>

                  <part_body>Hello</part_body>

            </part>

            <part>

                  <part_name>pdf</part_name>

                  <part_type>html</part_type>

                  <part_body>Hello</part_body>

            </part>

      </content>

      <footer/>

</message-config>
```

The recurring payment not-scheduled notification template is as follows:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!-- edited with XMLSPY v5 rel. 4 U (http://www.xmlspy.com) by Usha (edocs) -
->
```

```
<message-config xmlns="http://www.edocs.com/messaging"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.edocs.com/messaging

content.xsd">

      <header>

            <subject>Recurring Payment Not Scheduled</subject>

      </header>

      <content>

            <no>123</no>

            <type>html</type>

            <body><![CDATA[<html>

            Dear $recurringPayment.getPayerId(),<br>

            <br>

            #if($isPaymentScheduled)

              #if($isCheck)

                  This email is to inform you a check payment has been
scheduled automatically for you.<br>

            The check amount is $$payment.getAmount(). The pay date is
$messagingTemplateUtil.getFormattedDate($payment.getPayDate(),"MM/dd/yyyy").<
br>

              #end

              #if($isCCard)

                  This email is to inform you a credit card payment has been
scheduled automatically for you.<br>

            The credit card amount is $$payment.getAmount(). The pay date is
$messagingTemplateUtil.getFormattedDate($payment.getPayDate(),"MM/dd/yyyy").<
br>

              #end

                  You can update or cancel this transaction following this
URL http://www.oracle.com.<br>

            #end

            #if($isPaymentNotScheduled)

                  This email is to inform you that your recurring payment
scheduled on
$messagingTemplateUtil.getFormattedDate($recurringPayment.getNextPayDate(),"M
M/dd/yyyy")

            is not made as requested.<br>
```

```
             #if($isAmountNegative)
                  #set( $negativeAmount = $payment.getAmount() * -1 )
                  The bill showed an amount of -$$negativeAmount.<br>
             #end
                  Please contact your biller for details.<br>
            #end
            #if($isLessPayment)
                  This email is to inform you that the amount due
$$recurringPayment.getBillAmountDue() is
            more than the maximal amount, $$recurringPayment.getAmount(),
specified in the recurring payment.
            The bill is not paid by recurring payment.<br>
            #end
            #if($isAlreadyPaid)
                  The bill, due on
$messagingTemplateUtil.getFormattedDate($recurringPayment.getBillDueDate(),"M
M/dd/yyyy"), is not paid
            by recurring payment because it is already been paid.<br>
            #end
            #if($isLastRecurringPayment)
                  This is the last payment from the recurring payment.<br>
            #end
            #if($isRecurringPaymentCanceled)
                  This email is to inform you that your recurring payment
schedule has been deactivated,
            due to your account status, and no payments have been
scheduled.<br>
                  Please contact your biller for details.<br>
            #end
            #if($isUptoAmountExceeded)
                  Also, the amount due $$recurringPayment.getBillAmountDue()
is more than the maximal amount, $$recurringPayment.getAmount(),
            specified in the recurring payment. You still have an unpaid
balance of $$unpaidBalance.<br>
            #end
```

```
                  Number of payments made until now is 
$recurringPayment.getCurrNumPayment().<br>

            </html>

            ]]></body>

            <part>

                  <part_name>doc</part_name>

                  <part_type>html</part_type>

                  <part_body>Hello</part_body>

            </part>

            <part>

                  <part_name>pdf</part_name>

                  <part_type>html</part_type>

                  <part_body>Hello</part_body>

            </part>

      </content>

      <footer/>

</message-config>
```

The following recurring notification template variables are used for both recurring payment scheduled and recurring payment not-scheduled notifications:

Variable Name	**Type**	**Description**
`recurringPayment`	`IRecurringPayment`	Contains recurring payment information and current bill information paid by this recurring payment, when applicable. Bill information is null if the amount and pay date are both fixed.
`isPaymentScheduled`	Boolean	True if a payment has been scheduled.
`isCheck`	Boolean	True if the payment scheduled is a check.
`isCCard`	Boolean	True if the payment scheduled is a credit card.
`payment`	`IPaymentTransaction`	`ICheck` if `isCheck` is true or `ICreditCard` if `isCCard` is true. This is the payment being scheduled.

Variable Name	Type	Description
`isPaymentNotScheduled`	Boolean	True if the payment is not scheduled for some reason. Usually this is because a payment job plug-in rejected the payment based on a customer business rule.
`isLessPayment`	Boolean	True if the amount due is less than a certain amount, but the amount due is more than that. Notify the customer to pay manually.
`isAlreadyPaid`	Boolean	True when Payment finds a `DuplicateBillIdException` during the insertion of a payment into database.
`isLastRecurringPayment`	Boolean	True if this is the last payment.
`isRecurringPaymentCancelled`	Boolean	True if the recurring payment is cancelled. For example, if the payment account is cancelled. See the job configuration for details.

Payment Notification Template

This template controls the format of emails that are sent to the administrator by each job. The template file is *payment_job_notification.xml.vm*:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!-- edited with XMLSPY v5 rel. 4 U (http://www.xmlspy.com) by Usha (edocs) -
->

<message-config xmlns="http://www.edocs.com/messaging"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.edocs.com/messaging

content.xsd">

      <header>

            <subject>Payment Job Notification</subject>

      </header>

      <content>

            <no>123</no>

            <type>html</type>

            <body><![CDATA[<html>
            #if($isOK)
              $taskName was done without error at
$messagingTemplateUtil.getFormattedDate($currentTime,"MM/dd/yyyy
HH:mm:ss").<br>
            #else
              $taskName was done with an error at
$messagingTemplateUtil.getFormattedDate($currentTime,"MM/dd/yyyy
HH:mm:ss").<br>
            #end


            #if($skipSynchronization)
              <br>
              As Skip SynchronizationTask setting is set to YES, The
Synchronization Task was skipped<br>
            #end


            #if($recurringPmtSyncTask)
             <br>
             #if($isDone)
                  Job Name : $jobName<br>
                  Total Number of RecurringPayments to be synchronized :
$syncCount<br>
                  Total Number of RecurringPayments that are synchronized
Successfully : $syncSuccessCount<br>
                  Total Number of Recurring Payments that failed to
synchronize : $syncFailureCount<br>
             #else
                  Please look at the audit tables for detail.<br>
                  <br>
                  Job Name : $jobName<br>
                  Total Number of RecurringPayments to be synchronized :
$syncCount<br>
                  Total Number of RecurringPayments that are synchronized
Successfully : $syncSuccessCount<br>
                  Total Number of Recurring Payments that failed to
synchronize : $syncFailureCount<br>
             #end
            #end


            #if($recurringPmtSchedulerTask)
             <br>
             #if($isDone)
                  Job Name : $jobName<br>
                  Total Number of RecurringPayments to be scheduled :
$scheduleCount<br>
                  Total Number of RecurringPayments that are scheduled
Successfully : $scheduleSuccessCount<br>
                  Total Number of Recurring Payments that failed to be
scheduled : $scheduleFailureCount<br>
                 #if($isDecryptFailed)
                  <br>
                  Total Number of Recurring Payments cancelled due to
decryption failure : $CancelCount<br>
                 #end
             #else
                  Please look at the audit tables for detail.<br>
                  <br>
                  Job Name : $jobName<br>
                  Total Number of RecurringPayments to be scheduled  :
$scheduleCount<br>
                  Total Number of RecurringPayments that are scheduled
Successfully : $scheduleSuccessCount<br>
                  Total Number of Recurring Payments that failed to be
scheduled  : $scheduleFailureCount.<br>
                 #if($isDecryptFailed)
                  <br>
                  Total Number of Recurring Payments cancelled due to
decryption failure : $CancelCount<br>


                 #end


             #end
            #end


            #if($paymentReminderTask)
             #if($isDone)
                  <br>
                  Job Name : $jobName<br>
                  Total Number of Good Check Payment notifications : 
$goodCheckPaymentsCount<br>
                  Total Number of Check Payment notifications failed due to 
decryption failure : $badCheckPaymentsCount<br>
                  <br>
                  Total Number of Good CreditCard Payment notifications : 
$goodCCPaymentsCount<br>
                  Total Number of CreditCard Payment notifications failed due 
to decryption failure : $badCCPaymentsCount<br>
             #end
            #end


            #if($CreditCardExpNotifyTask)
             #if($isDone)
                  <br>
                  Job Name : $jobName<br>
                  Total Number of CreditCard expiration notifications to be 
processed : $ccexpNotifyCount<br>
                  Total Number of CreditCard expiration notifications that 
are processed Successfully : $ccexpNotifySuccessCount<br>
                  Total Number of CreditCard expiration notifications that 
are failed : $ccexpNotifyFailureCount<br>
                  <br>
                  Total Number of Good CreditCard notifications : 
$goodCCAccountCount<br>
                  Total Number of Bad CreditCard notifications  : 
$badCCAccountCount<br>
             #end
            #end
            #if($CheckSubmitTask)
             #if($isDone)
                  <br>
                  Job Name : $jobName<br>
                  #if($isHoliday)
                        This job was not run since today
($messagingTemplateUtil.getFormattedDate($todayDate,"MM/dd/yyyy")) is a
holiday.<br>
                  #else
                        While running the job, there were account decryption
failures.<br>
                  #end
             #end
            #end

            #if($SubmitEnrollTask)
             #if($isDone)
                  <br>
                  Job Name : $jobName<br>
                  #if($isHoliday)
                        This job was not run since today
($messagingTemplateUtil.getFormattedDate($todayDate,"MM/dd/yyyy")) is a
holiday.<br>
                  #end

                  #if($isDecryptFailed)
                        While running the job, there were account decryption
failures.<br>
                  #end
             #end
            #end

            #if($CreditCardSubmitTask)
             #if($isDone)
                  <br>
                  Job Name : $jobName<br>
                  #if($isDecryptFailed)
                        While running the job, there were account decryption
failures.<br>
                  #end
             #end
            #end
            </html>
            ]]></body>
            <part>
                  <part_name>doc</part_name>
                  <part_type>html</part_type>
                  <part_body>Hello</part_body>
            </part>
            <part>
                  <part_name>pdf</part_name>
                  <part_type>html</part_type>
                  <part_body>Hello</part_body>
            </part>
      </content>
      <footer/>
</message-config>
```

pmtRecurringPayment Variables

The recurring notification template variables for the **synchronization task** are:

Recurring Synch Variable	Type	Description
skipSynchronization	Boolean (true or false)	True enables the skip synchronization option.
recurringPmtSyncTask	Boolean (true or false)	True identifies this as the recurring payment task.
isDone	Boolean (true or false)	True indicates that the job is done.
jobName	String	The job name.
syncCount	int	Total number of accounts to be synchronized.

Recurring Synch Variable	**Type**	**Description**
syncSuccessCount	int	Successful number of synchronized accounts.
syncFailureCount	int	Number of failed of synchronized accounts.

Example:

```
#if($recurringPmtSyncTask)
 <br>
 #if($isDone)
      Job Name : $jobName<br>
      Total Number of RecurringPayments to be synchronized :  $syncCount<br>
      Total Number of RecurringPayments that are synchronized Successfully :
$syncSuccessCount<br>
      Total Number of Recurring Payments that failed to synchronize :
$syncFailureCount<br>
 #else
      Please look at the audit tables for detail.<br>
      <br>
      Job Name : $jobName<br>
      Total Number of RecurringPayments to be synchronized : $syncCount<br>
      Total Number of RecurringPayments that are synchronized Successfully :
$syncSuccessCount<br>
      Total Number of Recurring Payments that failed to synchronize :
$syncFailureCount<br>
 #end
#end
```

The recurring notification template variables for the **scheduler task** are:

Recurring Scheduler Variable	**Type**	**Description**
recurringPmtSchedulerTask	String	Identifies the scheduler task.
isDone	Boolean (true or false)	To identify the job had done.
jobName	String	To identify the job name.
scheduleCount	Int	Total number of accounts to be scheduled
scheduleSuccessCount	Int	Successful number of scheduled accounts

Recurring Scheduler Variable	**Type**	**Description**
scheduleFailureCount	Int	Failed number of scheduled accounts
CancelCount	Int	Cancelled number of scheduled accounts
isDecryptFailed	Boolean value (true or false)	To identify whether there was/were decryption failure/s

Example:

```
#if($recurringPmtSchedulerTask)

 <br>

 #if($isDone)

      Job Name : $jobName<br>

      Total Number of RecurringPayments to be scheduled : $scheduleCount<br>

      Total Number of RecurringPayments that are scheduled Successfully :
$scheduleSuccessCount<br>

      Total Number of Recurring Payments that failed to be scheduled :
$scheduleFailureCount<br>

     #if($isDecryptFailed)

      <br>

      Total Number of Recurring Payments cancelled due to decryption failure
: $CancelCount<br>

     #end

 #else

      Please look at the audit tables for detail.<br>

      <br>

      Job Name : $jobName<br>

      Total Number of RecurringPayments to be scheduled  : $scheduleCount<br>

      Total Number of RecurringPayments that are scheduled Successfully :
$scheduleSuccessCount<br>

      Total Number of Recurring Payments that failed to be scheduled  :
$scheduleFailureCount.<br>

     #if($isDecryptFailed)

      <br>

      Total Number of Recurring Payments cancelled due to decryption failure
: $CancelCount<br>
```

```
        #end


    #end
                 #end
```

pmtPaymentReminder Variables

Reminder Variable	**Type**	**Description**
paymentReminderTask	String	Identifies the payment reminder task.
isDone	Boolean (true or false)	Identifies the job is done.
jobName	String	Identifies the job name.
goodCheckPaymentsCount	Int	Number of successful check accounts.
badCheckPaymentsCount	Int	Number of failed check accounts
goodCCPaymentsCount	Int	Number of successful credit card accounts.
badCCPaymentsCount	int	Number of failed credit card accounts.

Example:

```
#if($paymentReminderTask)

 #if($isDone)

<br>

Job Name : $jobName<br>

Total Number of Good Check Payment notifications :
$goodCheckPaymentsCount<br>

Total Number of Check Payment notifications failed due to decryption failure
: $badCheckPaymentsCount<br>

<br>

Total Number of Good CreditCard Payment notifications :
$goodCCPaymentsCount<br>

Total Number of CreditCard Payment notifications failed due to decryption
failure : $badCCPaymentsCount<br>

 #end

#end
```

pmtCreditCardExpNotify Variables

CCExpNotify Variable	Type	Description
CreditCardExpNotifyTask	String	Identifies the credit card expiration notification task.
isDone	Boolean (true or false)	Identifies the job is done.
jobName	String	Identifies the job name.
ccexpNotifyCount	int	Total number of notifications to be made
ccexpNotifySuccessCount	int	Number of successful accounts.
ccexpNotifyFailureCount	int	Number of failed accounts.
goodCCAccountCount	int	Number of good credit card accounts (due to successful decryption).
badCCAccountCount	int	Number of bad credit card accounts (due to unsuccessful decryption).

Example:

```
#if($CreditCardExpNotifyTask)
 #if($isDone)
          <br>
          Job Name : $jobName<br>
          Total Number of CreditCard expiration notifications to be processed : $ccexpNotifyCount<br>
          Total Number of CreditCard expiration notifications that are processed Successfully :
$ccexpNotifySuccessCount<br>
          Total Number of CreditCard expiration notifications that are failed :
$ccexpNotifyFailureCount<br>
          <br>
          Total Number of Good CreditCard notifications : $goodCCAccountCount<br>
          Total Number of Bad CreditCard notifications  : $badCCAccountCount<br>
 #end
#end
```

pmtCheckSubmit Variables

Check Submit Variable	Type	Description
CheckSubmitTask	Boolean value (true or false)	Identifies the check submit task

Check Submit Variable	Type	Description
isDone	Boolean (true or false)	Identifies the job had done.
jobName	String	Identifies the job name.
isHoliday	Boolean value (true or false)	Identifies a holiday.
dateUtil	DateUtil object	Format of the expiration date.
isDecryptFailed	Boolean value (true or false)	Identifies whether there was/were decryption failure/s.

Example:

```
#if($CheckSubmitTask)
 #if($isDone)
      <br>
      Job Name : $jobName<br>
      #if($isHoliday)
            This job was not run since today
($messagingTemplateUtil.getFormattedDate($todayDate,"MM/dd/yyyy")) is a
holiday.<br>
      #else
            While running the job, there were account decryption
failures.<br>
      #end
 #end
#end
```

pmtSubmitEnroll

Submit Ernoll Variable	Type	Description
SubmitEnrollTask	String	Identifies the submit enroll task
isDone	Boolean (true or false)	Identifies the job had done.
jobName	String	Identifies the job name.
isHoliday	Boolean value (true or false)	Identifies a holiday.
isDecryptFailed	Boolean value (true or false)	Identifies whether there was/were decryption failure/s.
todayDate	Java.util.Date	A Date object to identify today.

Example:

```
#if($SubmitEnrollTask)

 #if($isDone)

      <br>

      Job Name : $jobName<br>

      #if($isHoliday)

      This job was not run since today
($messagingTemplateUtil.getFormattedDate($todayDate,"MM/dd/yyyy")) is a
holiday.<br>

      #end


      #if($isDecryptFailed)

      While running the job, there were account decryption failures.<br>

      #end

 #end

#end
```

Credit Card Expiration Notification Template

When a credit card is about to expire, email notification messages are generated from the template file *payment_ccaccount_ccexpired.xml.vm*:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!-- edited with XMLSPY v5 rel. 4 U (http://www.xmlspy.com) by Usha (edocs) -
->

<message-config xmlns="http://www.edocs.com/messaging"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.edocs.com/messaging

content.xsd">

      <header>

            <subject>Credit Card Expired</subject>

      </header>

      <content>

            <no>123</no>

            <type>html</type>

            <body><![CDATA[<html>
            Dear $account.getUserId():<br>
            <br>
            This email is to remind you that your credit card which has the
account number $account.getShorttenedAccNumber(),
            #if($accExpired)
            has expired in
$messagingTemplateUtil.getFormattedDate($account.getExpireDate(),"MMM yyyy").
            #else
            is about to expire in
$messagingTemplateUtil.getFormattedDate($account.getExpireDate(),"MMM yyyy").
            #end
            #set( $size = "C01" )
            <br>
            <br>
            Please login to the edocs Payment system and update the credit
card information.
            <br>
            <br>
            Thanks
            </html>
            ]]></body>
            <part>
                  <part_name>doc</part_name>
                  <part_type>html</part_type>
                  <part_body>Hello</part_body>
            </part>
            <part>
                  <part_name>pdf</part_name>
                  <part_type>html</part_type>
                  <part_body>Hello</part_body>
            </part>
      </content>
      <footer/>
</message-config>
```

The credit card expiration notification template variables are:

variable	Value type	Description
accExpired	Boolean value (true or false)	Identify whether the account is expired or not
account	ICreditCardAccount object	Object of ICreditCardAccount that has the information about the account
accountNumber	String	The expired account number

Understanding the Payment Template Engine

The payment templates provide a generic template mechanism based on Java reflection. The template engine generates custom text output based on the templates. Similar to JSP, the template engine replaces the special placeholders inserted into the text file with the values of Java objects. For more detailed API documentation, see the ePayment Manager JavaDoc included with the SDK.

The Template engine hosts a pool of objects in its context in the form of a hash table. You can refer to the variables in that context by their names. For example, there is a Check object whose name is “check”. You can refer to that object as: %check%. This means replace %check% with the string returned from `check.toString()`. This is true for all Java objects except *java.util.Date*, where `getTime()` is called and inserts a long value that is the number of milliseconds since January 1, 1970, 00:00:00 GMT. If a method returns void, then nothing will be printed out.

The content of the message consists of text plus resolved placeholders. Placeholders are Java variables, which are ePayment Manager hosted objects including their attributes and methods.

For more information about the Template class, see the ePayment Manager SDK JavaDoc.

All template variables must be enclosed by two %s. To escape '%', use '%%'. For example, "%%40" means "%40"

In addition to referring to variables, you can also access an object’s public fields and methods. The valid reference is: %name.field%, %name.method(param1, param2, ...)%, where each parameter to a method can be either of name, name.field or name.method(param1, param2, ,,,). The number of parameters is unlimited and an arbitrary level of method nesting is allowed (nesting means that a method's return value is used as a parameter when calling another method). For example, suppose there are two objects in contexts: "buf" which is a StringBuffer, and "str" which is a String. The following references are valid: %buf%, %buf.append(str)%, %buf.append(str.toString())%.

A static field or method can be accessed directly without instantiating an object. For example, java.lang.Integer has a static field called MIN_VALUE and a static method called parseInt. You can refer to them as %java.lang.Integer.MIN_VALUE% or %java.lang.Integer.parseInt(“12.34”)%.

All variables must be preset by calling putToContext on the Template class. Some variables are already set by ePayment Manager which you can use directly. But you can also put your own variables into the context:

```
%template.putToContext(“buf”, new java.lang.StringBuffer())%
```

This means to put a new StringBuffer object called "buf" into the template context. You can then refer to this object by its name:

```
%buf.append(“abc”)%
```

This appends “abc” to the end of the StringBuffer’s value.

The current ePayment Manager engine has some limitations. One is that it cannot do math operations, for example: x + y. You must call a Java method to do math operations. Another limitation is that it doesn’t allow you to concatenate method calls, for example: %variable.method().method() %. You must write your own Java method to do method concatenation.

Included with the ePayment Manager package, there are a few utility classes to help you overcome the weakness of payment template engine. These classes are:

```
com.edocs.payment.util.DecimalUtil
com.edocs.payment.util.DateUtil
com.edocs.payment.util.StringUtil.
```

One useful method in StringUtil is concat. It is declared and used as follows:

```
public static String concat(String s1, String s2, String s3)
%com.edocs.payment.util.StringUtil.concat(s1,s2,s3)%
```

Remember, you cannot do %s1.concat(s2).concat(s3)% inside a template, instead, you must call this function from template:

```
%com.edocs.payment.util.StringUtil.concat(s1,s2,s3)%.
```

Another useful method is format() from DateUtil class. This method helps format a Date object into different display formats. For example: %com.edocs.payment.util.DateUtil.format(“MMM dd, yyyy”, check.getPayDate())% formats a check’s pay date to display as “Jan 01, 2000”. For a complete list of possible date formats, please check the JDK document about java.text.SimpleDateFormat.

When writing customized Java code, we strongly recommend that you use static methods as frequently as possible, so you can call them directly from a template without creating an instance of that object first. For example, by default, the individual ID field of an ACH entry detail field is populated with the customer’s account number using `%check.getPayerAcctNumber()%`. The returned result is 16 bytes long, but the actual account number is 15 bytes, so you must truncate the retrieved value. The following steps describe how to create a java class to do truncation, and enable it in the Payment template engine:

1. Write a Java class:

```
package com.edocs.ps;
public class MyUtil {
  public static String truncate(String s){
     return s.substring(1);
  }
}
```

2. Compile the class and put it into *payment_custom.jar* of each EAR file, then re-deploy the EAR files.

3. You can now refer to this class in a template as follows:

```
%com.edocs.ps.MyUtil.truncate(check.getPayerAcctNumber())%
```

Customizing ACH Templates

The ACH records of interest are in File Header, Batch Header, Entry Detail for PPD, Addenda and return for PPD, Batch Trailer and File Trailer. ACH fields may be mandatory, required, or optional. The contents of mandatory fields are fixed and should not be customized. Required fields are usually defined by the receiving bank, and may be customized for different banks. Optional fields can be customized, also.

By default, *secCode* is set to WEB to be compliant with the ACH 2001 format. However, you can change the SEC code based on the requirements of a biller's bank by editing the *batchHeader_template.xml* file.

The following table is a list of some ACH fields. The ACH fields can be customized **upon a billers' request**. The pmtCheckSubmit jobs running date is referred to as **Today**.

Field Name	**Where**	**Description**
Company Descriptive Date	8th field in batch header, optional	Default set to Today; the date that pmtCheckSubmit is running.
Effective Entry Date	9th field in batch, required	The date when checks in the batches need to be cleared. This is a suggested date from ACH, but the actual date that checks are cleared may vary. All checks with the same pay date will be put into one batch. The effective entry date may not always be the pay date. The default setting for effective entry date is: If the pay date is tomorrow or earlier, then it is the earliest business date after today. If the pay date is after tomorrow, then it is the earliest business date after the pay date (including the pay date).
Individual ID	7th field in PPD entry detail, optional or required	By default set to the customer's account with the biller. Since this field is 15 bytes, **the length of customer's account must not exceed 15 bytes**. If the customer account is longer than 15 bytes, either the field will not be populated, or you must truncate this field using Java code or the Java classes provided by ePayment Manager.
Individual Name	8th field in PPD entry detail. Required	By default set to the check's payment ID. Payment ID is the primary key on the *check_payments* table. It can be used to map a returned check back to the one in Payment database.

The templates for ACH are actually XML files, which describe the format of each ACH record, such as the start position, length, etc. There are two sets of templates: one to generate ACH files, and another to parse ACH return files.

The first set of templates is used to generate ACH files. They are *fileHeader_template.xml*, *batchHeader_template.xml*, *entryDetail_template.xml*, *batchTrailer_template.xml* and*Trailer_template.xml*. When an ACH file is generated, check information is pulled from the database and then populated into the content of the XML files by replacing the template variables. The resulting XML file is transferred into an ACH file according to the format specified by the XML tags. The generic format of an XML tag is:

```
<amount pos="30" len="10" fmt="N" fract="2">%
```

Where:

`amount` is the name of the tag
`pos` is the start position
`len` is the length of the field
`fmt` is the format of the field
`fract` is the number of digits after decimal point if the fmt is "N" (numerical).

The tables below list the template variables that are predefined in the Payment template engine. These variables are used to populate the content of the templates.

The following template variables are used by all templates:

Global Variable Name	**Type**	**Description**
template	*com.edocs.util.template.Template*	The template engine.
stringUtil	*com.edocs.payment.util.StringUtil*	Makes calling the static methods of *StringUtil* easier. Instead of using: `%com.edocs.payment.util.StringUtil.concat("a","b","c")%` use: `%stringUtil.concat("a", "b", "c")%`
decimalUtil	*com.edocs.payment.util.DecimalUtil*	Provides decimal number manipulations.
dateUtil	*com.edocs.payment.util.DateUtil*	Provides date manipulation methods Also a calendar, which includes all US holidays.
batch	*com.edocs.payment.IPaymentBatch*	The payment summary report, which you can view through the Command Center.
config	*com.edocs.payment.config.IPaymentConfig*	Payment setting information.
attributeName	*com.edocs.payment.config.AttributeName*	Payment setting parameter names. Use it with the variable `config` to get payment setting information.

The following template variables are used by **File Header**:

Variable Name	**Type**	**Description**
fileCreateDate	java.util.Date	Creation date of the ACH file.
fileCreateTime	java.util.Date	Creation time of the ACH file.
fileIdModifier	java.lang.String	ACH file modifier, “A” to “Z” and “0” to “9”.

The following template variables are used by **Batch Header**:

Variable Name	**Type**	**Description**
curPayDate	java.util.Date	The pay date of checks in the batch. All the checks in the same batch have the same pay date.
companyDescData	String	From Payment Settings.
companyDescDate	Date	Defaults to Today. To use another date, you must call a static Java method.
batchNumber	int	Starts from “1”; identifies the batches in the ACH.
batchEffectiveEntryDate	Date	Identifies the batches in the ACH.

The following template variables are used by **Entry Detail**:

Variable Name	**Type**	**Description**
check	com.edocs. payment.ICheck	All check payment information, including the trace number.
addenda Record Indicator	int	Indicates whether there is addenda record for entry detail. 0=No; 1=Yes.

The following template variables are used by **Batch Trailer**:

Variable Name	**Type**	**Description**
batchEntryHash	String	See the ACH documentation.
batchEntryAddendaCount	int	Number of entries in the batch.
batchDebitAmount	String	Total debit amount in the batch.
batchCreditAmount	String	Always “0”.

Template variables used by **Batch Trailer**:

Variable Name	Type	Description
batchCount	int	Number of batches in the file.
blockCount	int	See the ACH documentation.
totalEntryHash	String	See the ACH documentation.
totalEntryAddendaCount	int	Total number of entries in the file
totalDebitAmount	String	Total debit amount in the file.

Matching a Check in the ACH Return to the Database

Return files are parsed by the return templates, *fileHeader_return_template.xml*, *batchHeader_return_template.xml*, *entryDetail_return_template.xml*, *addenda_return_template.xml*, *batchTrailer_return_template.xml* and *fileTrailer_return_template.xml*. The format of these files is similar to the format of the submit templates described previously. For example:

```
<individualName pos="55" len="22" fmt="AN"
target="%check.setPaymentId(?)%"></individualName>
```

retrieves the part of the text from positions 55 to 77, puts them into a variable called "?" and then calls `check.setPaymentId()` to set *payment_id* for the check. The template executes the template statement specified by XML tag "target" only.

When a check is returned from the ACH network, Payment matches it to that check in the database and marks it as returned. ACH modifies several fields in the return file. Payment populates one or more unchanged fields with identification information to help in matching them back to a check in the database. Consult the ACH documentation for information about which fields are not changed.

The return template does two things. First, it retrieves the error return code from the addenda record, and then tries to reconstruct the payment ID or gateway payment ID to match a check in the database. If Payment cannot populate the payment ID into the ACH file, it uses the gateway payment ID, which is a concatenation of a few check payment fields that can identify a check. The procedure is described in the following steps:

By default, Payment populates the *payment_id* of the check into the individual name field to create the ACH file. The following line in *entryDetail_template.xml* populates the payment ID into an individual name:

```
<individualName pos="55" len="22"
fmt="AN">%check.getPaymentId()%</individualName>
```

The following line in *entryDetail_return_template.xml* extracts the payment ID:

```
< individualName pos="55" len="22" fmt="AN"
target="%check.setPaymentId(?)%"></individualName >
```

The following line in *addenda_return_template.xml* extracts the return error code:

```
<returnReasonCode pos="4" len="3"
target="%check.setTxnErrMsg(?)%"></returnReasonCode>
```

Payment then changes the status of the check to "returned" and updates this check in the database using its *payment_id*.

If the individual name is required for something else, for example the check account name (which is the first 22 bytes), then following these steps to use gateway payment ID:

1. Modify *entryDetail_template.xml* to populate individual name with account name. Change:

```
<individualName pos="55" len="22"
fmt="AN">%check.getPaymentId()%</individualName>
```

To:

```
<individualName pos="55" len="22"
fmt="AN">%stringUtil.substring(check.getAccountName(), 0,
22)%</individualName>
```

2. Modify *entryDetail_return_template.xml* so that payment ID won't be set for a returned check. Change:

```
<individualName pos="55" len="22" fmt="AN"
target='%check.setPaymenId(?)%'></individualName>
```

To:

```
<individualName pos="55" len="22" fmt="AN"></individualName>
```

3. Since payment ID cannot be used to match checks, we can use gateway payment ID instead. Gateway payment ID is the ID generated by the template that submitted the ACH file to ACH. This template generates a unique ID based on the information submitted to ACH. This ID must contain information that won't be changed by ACH in the return file. The Payment engine will use the gateway payment ID to find a match in the database.

In very rare circumstances, more than one match may be found. In that case, the match with the latest creation time is used. The following example discusses several ways to generate the gateway payment ID.

ePayment Manager generates a trace number and puts that into the entry detail record. By default, the trace number starts at 0000000 and increases by one for each check until it reaches 9999999. After this point, the numbering restarts at 0000000. It's possible to get a duplicate trace number (after 10 million checks). However, since the ePayment Manager engine always chooses the payment with the latest date, the correct check will be matched. You can use both the trace number and individual ID (customer account number) to identify a payment and use them for the gateway payment ID.

Example 1: Unchanged ACH trace number

In the following example, we assume that the ACH/Bank will return both original trace number and individual ID to ePayment Manager. To do that:

1. At the start of *entryDetail_template.xml*, see the section:

```
<ACH_6>
%<*>%
%check.setGatewayPaymentId(com.edocs.payment.util.StringUtil.c
oncat(check.getPayerAcctNumber(), "_", check.getTxnNumber()))%

%</*>%
```

This statement is commented out in the template, using %<*>% and %</*>%. Removing the comment tags enables the statement.

The trace number is stored as *txnNumber* in the check object. This statement concatenates the customer account number, a "_", and trace number as the gateway payment ID. The `setGatewayPaymentId` method returns void, so nothing will print out. (If it did return a value, then that would print, which would ruin the format of the XML file.) After running pmtCheckSubmit, check the gateway payment ID in the *check_payments* table, which should be the concatenation of the individual ID and the trace number that are written into the entry detail record.

2. Next, ePayment Manager retrieves the original trace number from the return file, and sets it as the gateway payment ID. In the *addenda_return_template.xm*, find this section:

```
<traceNumber pos="80" len="15" fmt="N"
target1='%check.setGatewayPaymentId(txnNumber)%'
target2='%check.setGatewayPaymentId(stringUtil.concat(payerAcc
tNumber, "_", txnNumber))%'></traceNumber>
```

Rename "target2" to "target", which will reconstruct the gateway payment ID based on the returned customer account number and trace number. Template variable `payerAcctNumber` has been set in *entryDetail_return_template.xml* and *txnNumber* has been set before this line in the *addenda_return_template.xml* by calling *template.putToContext*.

3. Now you are all set. You should test this setting using an actual return file and verify that the check's *status* has been updated to –4 in the *check_payments* table.

Example 2: Modified ACH trace number

If the individual ID is not returned as it was set, you can try to use other information, such as individual name combined with trace number. If only the trace number can be used for gateway payment ID, use that by:

1. At the start of *entryDetail_template.xml*, see the section:

```
        %<*/>%
%check.setGatewayPaymentId(check.getTxnNumber())%
%</*>%
```

Remove the comment tags to enable the statement.

2. In addenda_return_template.xml , see the section:

```
<traceNumber pos="80" len="15" fmt="N"
target1='%check.setGatewayPaymentId(txnNumber)%'
target2='%check.setGatewayPaymentId(stringUtil.concat(payerAcc
tNumber, "_", txnNumber))%'></traceNumber>
```

Rename "target1" to "target" to enable using trace number as gateway payment ID.

Using achReturn_wl

achReturn_wl.sh (on Windows, it is achReturn_wl.bat) generates ACH return files. You need to take a look at this script and make necessary change to it based on your install environment.

First, edit the script and make sure EDX_HOME, PAYMENT_HOME, WL_HOME, WL_HOST and WL_PORT are set correctly.

Second, to run *achReturn_wl.sh*:

```
achReturn_wl.sh -payeeId payee
    -returnParams payment_id return_reason(3 chars)
    -nocParams payment_id noc_code addenda_info
    -fileCreateDate(yyMMdd) date -fileIdModifier id
```

The arguments to *achReturn_wl.sh* are described in the following table:

Argument and Parameters	**Description**
-payeeId payee	The DDN name. Required.
-returnParams payment_id return_reason	Generates returns for a check.payment_id is the payment ID of a regular check or prenote check for which you want to generate a return.return_reason is a three-character string starting with R. See the ACH specification for a list of return codes.
-nocParams payment_id noc_code addenda_info	Generates NOC for a check.payment_id is the payment ID of a regular check or a prenote for which you want to generate a NOC.noc_code is a three-character string starting with C.addenda_info is the correct payment account data. It must satisfy the format required by the corresponding NOC code.
-fileCreateDate date	The date when the file was created by ACH. This is optional and usually not required.
-fileIdModifier id	id is the file ID modifier for the ACH file. This is optional and usually not required.

The generated ACH return file is put into the file input directory defined in Payment Settings of the Command Center.

addenda_info may include a space, and space is interpreted as a parameter separator instead of part of the addenda info. For example, for C03, *addenda_info contains*:

```
"01100390   134985425"
```

There are three spaces between the routing number and the account number. In this case, you can pass in "+" in place of " ", for example, "01100390+++134985425".

To generate a regular check return:

1. Make a new check payment from the UI with the pay date as today. Find the payment ID of this check in the *check_payments* table.

2. Run pmtCheckSubmit to submit it. The check status will change to processed. If you are using gateway payment ID, you will see that field is populated with the information as defined by the template.

3. Run the *achReturn_wl* utility passing the *–returnParams* option to generate an ACH return file. Be sure to run it from WebLogic installation directory.

4. Run pmtCheckUpdate to read the return file. After processing, the file is moved to the history directory, and the check status is changed to "returned".

5. View the exception report from Command Center.

6. If your check status has not changed, make sure the payee ID matches the DDN specified in the pmtCheckUpdate job. Also make sure that the payment gateway ID is configured correctly (if you are using the gateway payment ID).

To generate a prenote return:

1. Enroll a new customer from the UI. The customer status should be "pnd_active".

2. Run pmtSubmitEnroll to create a prenote file. A zero amount check will be inserted into *check_payments* table with a status of "prenote_processed". Find the check and write down its payment ID. If you are using the gateway payment ID, you will see that field is populated with the information as defined by the template.

3. Run the achReturn_wl utility passing the –returnParams option to generate an ACH return file. Be sure to run it from WebLogic installation directory.

4. Run pmtCheckUpdate to read that file, after processing, the file will be moved to the history directory, and the prenote check status is changed to prenote_returned, and the account_status field is changed to bad_active.

5. View the exception report from Command Center, where you can see the ACH prenote check.

If account status has not changed, make sure the payee ID matches the DDN specified by the pmtCheckUpdate job. Also make sure the payment gateway ID is configured correctly (if you are using the gateway payment ID).

To generate a NOC return:

1. Make a new check payment from the UI with a pay date of today. Find the payment ID of this check in the *check_payments* table. Do not use an existing payment ID for NOC testing.

2. Run pmtCheckSubmit to submit the check. The check status changes to "processed".

3. Run the *achReturn_wl* utility passing `–nocParams` to generate an ACH return file. Be sure to run it from the WebLogic installation directory.

You must calculate the addenda info so that it is formatted as the ACH spec requires. For example:

NOC code	**addenda info example**
C01	134985425
C02	01100390
C03	01100390 134985425
C05(saving)	37
C05(checking)	26
C06	134985425 37

NOC code	**addenda info example**
C07	01100390134985425 37

Pay careful attention to the number of spaces in the addenda info for C03, C06 and C07. Since there are spaces in the addenda info, you need to escape them.

4. Run pmtCheckUpdate to read the return file. After processing, the return file is moved to the history directory. A new check with zero amount and status of "noc_returned" is inserted into the *check_payments* table. The payment account information in the *payment_accounts* table is updated to the new account information, and the *txn_message* column records the NOC code, the new acct information and the old account information. *notify_status* is updated to "N".

5. View the exception report from the Command Center where you can see the new NOC check

6. If account information has not changed, make sure the payee ID matches the DDN of the pmtCheckUpdate job. Also make sure the payment gateway ID is configured correctly (if you are using a gateway payment ID).

To generate a return file with multiple check returns, prenote returns and/or NOC returns, repeat the `–returnParams` and `–nocParams` options.

To split (break) an ACH file into lines of 94 bytes:

1. Set the classpath the same as the one set inside *achReturn_wl.sh*.

2. Run the following command from the directory where the ACH file is:

```
java com.edocs.payment.cassette.ach.test.AddNewLines –achFile achFileName >
new_file_name
```

The previous command is not tied to a particular application server, so it can be used for both WebLogic and WebSphere.

To use achReturn_ws.sh for WebSphere:

On WebSphere, use *achReturn_ws.sh* to generate return files. The command line options are the same as the *achReturn_wl.sh*, but the deployment process is different.

WebSphere requires that the client application be packaged as an EAR file, and cannot be directly invoked. The EAR file is called *ear-ach-return.ear*, which you must assemble and deploy it before you can use it.

1. Assemble *ear-ach-return.ear*. Remember to put payment_*client.jar*, *payment_common.jar* and *payment_custom.jar* on the class path of the assembly tool.

2. Deploy *ear-ach-return.ear* on the same application server where *ear-eStatement.ear* is deployed.

3. Modify *achReturn_ws.sh* to reflect your application environment. For example, you may need to modify PAYMENT_HOME.

4. Run the script to generate returns.

5. You can also use WebLogic to generate ACH return files even though you are working on WebSphere. Points the WebLogic script to the same database used by WebSphere, then *run achReturn_wl.sh* as described previously.

Customizing the ACH Prenote File

To enroll a new customer with an ACH payment gateway, an ACH prenote file must be sent to the payment gateway to verify the customer's check account number and routing number. In most cases, the ACH prenote file format does not need to be customized. This section describes how ePayment Manager processes prenote files.

As of ePayment Manager 1.2, the prenote file is generated using the same set of template files that are used for regular check submission. For each payment account that requires a prenote, a zero amount check is inserted into the *check_payments* table, and information from that check is used to create the entry detail records. When a prenote is returned, the prenote check is located in the *check_payments* database using either payment ID or gateway payment ID, and the corresponding payment account is located in the *payment_accounts* table using *payer_id* (user ID) and check account number.

A prenote can also be returned.

Upper Case in an ACH File

The ACH specification is vague about whether lower case letters are allowed. If upper case is required, lower case **must** be converted to upper case. You can do this by editing the templates, or customizing the UI.

To support upper case, edit the templates to make sure the ACH payment setting parameter values are all upper case. Then, convert the individual ID and individual name fields in the entry detail record (*entryDetail_template.xml*). You can use the `toUpperCase` method from *com.edocs.payment.util.StringUtil* to translate lower case to upper case. The case of the other fields in the entry detail record does not matter.

To customize the enrollment UI, convert the information in the individual name and individual ID fields to upper case before saving them in the database.

Customizing CheckFree CDP Files

Checkfree CDP files are well defined, and normally do not require customization. Checkfree does not change the payment ID, so it is easy to match checks returned by Checkfree to the ePayment Manager database.

To generating Checkfree response files for testing:

A Checkfree response file is generated based on a payment file sent to the check file. Response files include confirm files and journal files.

cdpOut2In_wl.sh generates Checkfree CDP response files. You must run this script in the directory where you installed WebLogic. You may also need to change the following environment variables:

- EDX_HOME, if eStatement Manager is not installed on */opt/eStatement*
- PAYMENT_HOME, if Payment is not installed on */opt/ePayment*
- WL_HOME, if WebLogic is not installed in */opt/bea/weblogic92*

- WL_HOST, if WebLogic is not running on the local host
- WL_PORT, if WebLogic is not running on port 7001.

To run *cdpOut2In_wl.sh*:

```
cdpOut2In_wl.sh -payeeId payeeId
-cdpFile originalCdpFile
-outputDir output_dir_for_the_generated_files
-confirmParams confirmFileStatus(ACCEPTED/REJECTED) errCode(5
digits) errMsg
-journalParams 4000_record_index(starts from 0) checkfreeId
transactionStatus(ACCEPTED/REJECTED) errCode(5 digits) errMsg
```

The arguments to *cdpOut2In_wl.sh* are described in the following table:

Argument and Parameters	**Description**
-payeeId payeeId	DDN name
-cdpFile originalCdpFile	The name of CDP file sent to Checkfree. The confirm file and journal file are based on this file.
-outputDir output_dir_for_the_generated_files	The generated confirm file and journal file are put into this directory.
-confirmParams confirmFileStatus(ACCEPTED/REJECTED) errCode(5 digits) errMsg	Specifies the parameters for confirm files, separated by spaces. confirmFileStatus is either ACCEPTED or REJECTED. errCode is the five-digit Checkfree error code. errMsg is the error message. Enclose errMsg in double quotes if it includes spaces.
-journalParams 4000_record_index(starts from 0) checkfreeId transactionStatus(ACCEPTED/REJECTED) errCode(5 digits) errMsg	Specifies the parameters for journal files. A journal file is generated based on the 4000 records in the submitted CDP file. The first parameter specifies the index of the 4000 records in the submitted CDP file. The index starts from 0, with the first 400 records as 0. `checkfreeid` is the Checkfree ID, which can be anything, as long as the length is within the ACH specification. `transactionStatus` is either ACCEPTED or REJECTED. errCode is 5 digit Checkfree error code. errMsg is the error message.

Repeat `–journalParameters` for any other 4000 records in the submitted CDP file, if you want to generate responses for those records.

To generate Checkfree Return files for testing

A Checkfree return file is generated when a check is returned for various reasons.

cdpReturn_wl.sh generates Checkfree CDP return files. You must run this script in the directory where you installed WebLogic. You may also need to change the following environment variables:

- EDX_HOME, if eStatement Manager is not installed on /opt/eStatement
- PAYMENT_HOME, if ePayment Manager is not installed on /opt/ePayment
- WL_HOME, if WebLogic is not installed in /opt/bea/wlserver6.0

- WL_HOST, if WebLogic is not running on the local host
- WL_PORT, if WebLogic is not running on port 7001.

To run *cdpReturn_wl.sh*:

```
cdpReturn_wl.sh -payeeId payee
  -fileCreateDate YYYYMMDD
  -fileCreateTime HHMMSSmmmmmm
  -returnParams payment_id return_reason(3 chars)
debit_trace_number(7 digits)
  -debitEffectiveDate(MM/dd/yyyy) debit_effective_date -
debitReturnDate return_date
```

The arguments to *cdpReturn_wl.sh* are described in the following table:

Argument and Parameters	**Description**
-payeeId payee	The DDN name.
-fileCreateDate YYYYMMDD	Date when the return file will be marked as generated.
-fileCreateTime HHMMSSmmmmmm	Time when the return file will be marked as generated.
-returnParams payment_id return_reason(3 chars) debit_trace_number(7 digits)	Specifies the return parameters. payment_id return is the check payment ID that you want to generate a return for. return_reason is the 3 char ACH return code. debit_trace_number is the ACH trace number for that check.
-debitEffectiveDate debit_effective_date (MM/dd/yyyy)	Debit effective date. Optional.
-debitReturnDate return_date	Debit return date. Optional.

8 Generating Accounts Receivables (A/R) Files

Overview

It is often necessary to synchronize the ePayment Manager system with a biller's A/R system. ePayment Manager usually needs to periodically send A/R files to a biller's A/R system, which includes the payments being made through ePayment Manager. The format of the file varies among billers. To support this function, ePayment Manager has the pmtARIntegrator job, which uses a template and XML/XSLT to generate output in a variety of file formats.

The pmtARIntegrator job queries the ePayment Manager database to get proper payments, and then writes the payments into a flat file or an XML file using the Payment Template engine. The XML file can be further transformed into other format by using XSLT. The default implementation of this job does following things:

1. Queries the Payment database to get a list of check and/or credit card payments. The query is defined in *arQuery.xml* file, which finds all the check and credit card payments where the *payee_id* matches the current job DDN , the *status* is 8 ("paid") and *arflag* is "N".

2. Invokes the `process()` method of the default implementation of *com.edocs.payment.tasks.ar.IARPaymentIntegrator*, which is *com.edocs.payment.tasks.ar.SampleARPaymentIntegrator*. In this method, `ARPaymentIntegrator` writes the payments into a flat file or XML file using the Payment Template engine. There are two templates provided by Payment:

   - *arFlat_template.txt, which generates a flat A/R file*
   - *arXML_template.xml, which generates an XML file*

   The output file name is: *ar_yyyyMMddHHmmssSSS.extension*, where *extension* matches the extension of the template file.

3. Inside the `process()` method, if the output is an XML file, `SampleARPaymentIntegrator` can optionally apply an XSLT file against the output file to transform it into another format. The transformed file name is: *ar_trans_yyyyMMddHHmmssSSS.extention*, where *extension* is defined by the pmtARIntegrator job configuration.

4. Inside the `process()` method, `SampleARPaymentIntegrator` updates *arflag* of both check and credit card payments to "Y", and writes that to database. This ensures these payments won't be processed again by the next run of pmtARIntegrator.

Customizing arQuery.xml

The SQL queries used by the pmtARIntegrator job are defined in an XML file, *arQuery.xml*, which is provided by the default Payment installation. *arQuery.xml* is based on eStatement Manager XMLQuery technology. For details about this definition, see the *SDK Guide for Oracle Siebel eStatement Manager*.

CAUTION: eStatement Manager XMLQuery supports paging, but this feature **must** not be used for this job.

Most of the A/R file creation is done by an implementation class of the interface *com.edocs.payment.tasks.ar.IARPaymentIntegrator*. This adaptor interface provides maximum flexibility for customizing this job. The default implementation is *com.edocs.payment.tasks.ar.SampleARPaymentIntegrator*.

Before the actual query is executed in the database, the job invokes the `getMap()` method of `IARPaymentIntegrator`, which gets a list of objects that are used to replace the variables "?" defined in the SQL query of *arQuery.xml*. See the ePayment Manager SDK JavaDoc about *IARPaymentIntegrator* for more information.

The default `IARPaymentIntegrator` implementation, *SampleARPaymentIntegrator*, uses this *arQuery.xml* for database query:

```
<?xml version="1.0" encoding="UTF-8"?>
<query-spec>
  <data_source_type>SQL</data_source_type>

<query name="checkQuery">
    <sql-stmt><![CDATA[select * from check_payments where
payee_id = ? and statu
s = 8 ]]></sql-stmt>
    <param name="payee_id" type="java.lang.Integer"
position="1"/>
    <!--param name="last_modify_time"
type="java.sql.Timestamp" position="2" /-->
  </query>

  <query name="creditCardQuery">
  <sql-stmt><![CDATA[select * from creditcard_payments where
payee_id = ? and st
atus  = 8 and arflag = 'N']]></sql-stmt>
  <param name="payee_id" type="java.lang.Integer"
position="1"/>
  </query>

</query-spec>
```

Two queries are defined:

- **checkQuery** - Queries check payments
- **creditCardQuery** - Queries credit card payments

Both these queries get all the successful payments (*status*=8) of the current payee (biller or DDN of current job) from the relevant ePayment Manager tables. They both use *arflag* as a flag to prevent a payment from being sent to the A/R job twice. This flag is initially set to "N" when the payment is created. After the A/R job runs, the `SampleARPaymentIntegrator` changes the flag to "Y".

When using *arflag* as an A/R flag, you can create an index for it to increase performance. ePayment Manager provides a script just for that purpose in *ePayment/db/<DB_Type>/create_ar_index.sql. (DB Type refers to Oracle, DB2, MSSQL, etc.)* This script is not run when the ePayment Manager database is created, so you must run it manually.

Each of the queries in *arQuery.xml* has an SQL variable ('?') that must be resolved before the query can be sent to the database. The A/R job calls the `getMap()` method of `IARPaymentIntegrator` to get a Map of query variables, and uses their values to replace the '?' variables in the query. The names of the Map elements should match those defined in the "param" tags of the "query" tags.

For example, the default *arQuery.xml* has the "param" tag:

```
<param name="payee_id" type="java.lang.Integer" position="1"/>
```

To support this you should define a Map element whose name is "payee_id" and whose value (which must be an Integer, and contains the DDN reference number) replaces the "?" mark with "payee_id" in the query:

```
select * from check_payments where payee_id = ? and status = 8
and arflag = 'N'
```

The following query result set will be transferred to a list of checks (`ICheck` objects) for checkQuery, and credit cards (`ICreditCard` objects) for creditCardQuery, and then pass that list to the `process()` method of `IARPaymentIntegrator`.

CAUTION: The eStatement Manager `XMLQuery` object supports paging, but this feature **must not** be used for A/R query.

You can modify this file to use different queries.

Query Case Study

The new requirement for this example is to retrieve all payments whose status is returned or paid between 5:00PM today (the job run date) and 5:00PM yesterday (yesterday's job run date).

Step 1

Change *arQuery.xml* for checkQuery:

```
<query name="checkQuery">
<sql-stmt><![CDATA[select * from check_payments where
payee_id=? and status in (8,-4) and last_modify_time >= ? and
last_modify_time < ? ]] </sql-stmt>
<param name="payee_id" type="java.lang.Integer" position="1"/>
<param name="min_last_modify_time" type="java.sql.Timestamp"
position="2"/>
<param name="max_last_modify_time" type="java.sql.Timestamp"
position="3"/>
</query>
```

TIP: Use `java.sql.Timestamp` instead of `java.util.Date`.

Step 2

Do the same thing for creditCardQuery:

1. Since you are adding more "?"s to the query, you need to override the `getMap()` method of the default `ARPaymentIntegrator`:

```
pacakge com.edocs.ps.ar;
import java.util.*;
import com.edocs.payment.util.DateUtil;

public class MyARIntegrator extends ARPaymentIntegrator

{

   /**Override this method to populate the SQL variables in 
arQuery.xml

    */

public Map getMap(ARPaymentIntegratorParams 
payIntegratorParam,
                                 String objectFlag) throws 
Exception
{
        //call super class because we need to get the payee_id 
value
        Map map = super.getMap(payIntegratorParam, 
objectFlag);
        //no need to check objectFlag because we actually 
populate the
        //same values for both checkQuery and creditCardQuery
        Date today = new Date();

        today = DateUtil.dayStart(today);//set to 00:00:00AM
        Date today5 = DateUtil.addHours(today, 17); //set to 
05:00:00PM

        Date yesterday5 = DateUtil.addHours(today, -7) ;//set 
to 05:00:00PM of yesterday
        map.put("min_last_modify_time", 
DateUtil.toSqlTimestamp(yesterday5));

        map.put("max_last_modify_time", 
DateUtil.toSqlTimestamp(today5));
}


}
```

2. If you wish to make the cutoff time configurable instead of fixed at 5:00PM, use the flexible configuration fields of the A/R job, which are passed in as part of `ARPaymentIntegratorParams`. For more information about `ARPaymentIntegratorParams`, see the ePayment Manager SDK JavaDoc.

3. Compile your class using the *payment_client.jar* and *payment_common.jar* that comes with Payment, package the compiled class into the ePayment Manager EAR files, and re-deploy the EAR files.

4. Login to the Command Center and change the configuration of the A/R job to use the new implementation of the `IARPaymentIntegrator`, *com.edocs.ps.ar.MyARIntegrator*.

Customizing arFlat_template.txt

Payments returned by *arQuery.xml* are written to an A/R file using a Payment template file. Two templates come with ePayment Manager:

arFlat_template.txt- generates a flat A/R file

arXML_template.xml - generates an XML A/R file

arFlat_template.txt generates a sample flat A/R file. If this file includes most of your required data, but the format is not what you want, you can edit the template file to generate your own format. For more information about using the Template class, see the ePayment Manager JavaDoc.

The A/R job using *arFlat_template.txt* does two things:

1. Loops through the list of check and credit card payments to print out their details.

2. Calculates the totals for check debits, check credits, credit card debits and credit card credits (reversals).

Customizing arXML_template.xml

arXML_template.xml generates the same information as *arFlat_template.txt*, but in XML format. After creating the XML file, you can use XSLT to transform it into another XML file or into a flat file. The default *arTransform.xsl* transforms the original XML file into the same format as the one generated by *arFlat_template.txt*. Using XSLT is the recommended way to do the customization, because it is easy and powerful.

The A/R job using *arXML_template.xml* does two things:

1. Loops through the list of check and credit card payments to print out their details.

2. Calculates the totals for check debits, check credits, credit card debits and credit card credits (reversals).

To generate different file formats, change *arTransform.xsl*. Or, customize the *arXML_template.xml* file directly.

Customize arXML_template.xml and Use XSLT to Generate XML/Flat AR File

The *arXML_template.xml* generates the same information as *arFlat_template.txt*, but in XML format. After generating the XML file, you can use XSLT to transfer it into another XML file or into a flat file. The default *arTransform.xsl* transforms the XML file into the same format as the one generated by *arFlat_template.txt*. If you are familiar with XSLT, this is the recommended way to do the customization because XSLT is easy to use and powerful.

This template does two things:

1. Loops through the list of check and credit card payments to print out their details.

2. Calculates the totals for check debits, check credits, credit card debits and credit card credits (reversals).

To generate different file formats, change *arTransform.xsl*. If required, you can also customize the *arXML_template.xml* file.

To rename the generated files:

To rename the files generated by these utilities you must write a simple implementation of `IARPaymentIntegrator`. The following code demonstrates how to rename the XSLT output file to another name:

```
import java.io.*;
public class MyARIntegrator extends ARPaymentIntegrator
{
protected void
getTransformedARFileName(ARPaymentIntegratorParams
               payIntegratorParam, ) throws Exception
{
return ”newARName.txt”;
}
}
```

Re-implement IARPaymentIntegrator

You may want to re-implement the default `SampleARPaymentIntegrator` if you wish to add any of the following features. The following steps describe how to do this:

1. Re-name the default AR files.

2. Change the SQL query to add more “?” variables and to set values for those variables in the IARPaymentIntegrator implementation.

3. Add any additional steps, such as putting more objects into Template context before it is parsed.

4. Change the result of the template parsing. For example, because of limitations of Template engine, sometimes unwanted empty new lines are added. You should remove those lines.

5. Modify the check or credit card objects before they are updated in the database. By default, only `arflag` is updated from "N" to "Y". Another alternative is to update the check or credit card object in the template, and all your updates will be updated in the database.

To add any of the preceding features, you must extend from *SampleARPaymentIntegrator* and configure the pmtARIntegrator job to use your implementation.

You can overwrite following methods for your customization:

1. `getARFileName()`: overwrite to change the name of the AR flat file generated from arFlat_template.txt.

2. `getMap()`: overwrite

Select Only Check or Credit Card Payments

A biller may support only one of check or credit card payments. In this case, you must configure the pmtARIntegrator job to leave the “Credit card query name in XML query file” field blank. Also, you may want to customize the template files (*arFlat_template.txt* or *arXML_template.xml*) to remove any reference to the unavailable payment type, but this is optional.

Compiling and Packaging a Custom IARIntegrator

If you re-implement `IARIntegrator` or you have some custom Java classes to call from the AR template, you must re-compile and package your changes.

In most cases, you put your custom code into *payment_custom.jar*. Unfortunately, the `IARIntegrator` and its related classes are packaged as part of *ejb-payment-ar.jar*, not *payment_custom.jar*, so a different procedure is required.

To compile, you may need to put *ejb-payment-ar.jar* along with *payment_common.jar*, *payment_custom.jar* and *payment_client.jar* in your class path to re-implement IARIntegrator.

To package, drop all your AR custom classes into the *ejb-payment-ar.jar*.

A/R Filenames

The generated A/R files have default names of *ar_yyyyMMddHHmmssSSS.template_file_ext*, where the *template_file_ext* is the file extension of the template file. The XSLT transformed file has default name of *ar_trans_yyyyMMddHHmmssSSS.extension*, where *extension* is defined by the pmtARIntegrator job configuration. You may want to rename these files to a more meaningful name.

To rename the files, write a simple implementation of `IARPaymentIntegrator`. The following code demonstrates how to rename the XSLT output file to another name:

```
package com.edocs.ps.ar;

import com.edocs.payment.tasks.ar.*;

public class MyARIntegrator extends ARPaymentIntegrator

{

/**Override this method to give a new name*/

protected void
getTransformedARFileName(ARPaymentIntegratorParams
               payIntegratorParam, ) throws Exception
{
```

```
return ”newARName.txt”;
}
}
```

Single Payment Type

A biller may have only ACH and not credit card payments, or vice versa. In this case, you can customize the template files (*arFlat_template.txt* or *arXML_template.xml*) to remove any references to the unavailable payment type.

Or, when configuring the pmtARIntegrator job enter an empty value for the `Check query name in XML query file` or `Credit card query name in XML query file` parameter.

9 Packaging ePayment Manager Custom Code

Overview

You can package your custom code, both plug-in code and custom A/R jobs and templates, by adding it to *payment_custom.jar*. The ePayment Manager EAR files will access this JAR, and find the custom code. The ePayment Manager EAR files are merged into the eStatement Manager EAR file as part of installation, so your custom code will also be seen by the eStatement Manager Command Center.

To make this JAR file accessible by all the ePayment Manager EJB, JAR and WAR files, place it in the classpath of the *MANIFEST* file of each JAR and WAR file. For details of how the *MANIFEST* file works, refer to the J2EE or EJB specifications or the *Deploying and Customizing J2EE Applications Guide for Oracle Siebel eStatement Manager*. When the EJB JAR or WAR files are loaded, this JAR will be loaded and can be accessed by the EJB jar files or war files.

CAUTION: Never put your custom EJB code into *payment_custom.jar*; put your EJB code in your own JAR files.

To write a new plug-in for `IAchCheckSubmitPlugIn`:

1. Write and then compile your implementation class. You may want to use *payment_common.jar* and *payment_client.jar* from Payment as part of your class path.

2. Create a JAR file called *payment_custom.jar*, or use the *payment_custom.jar* from any of the Payment EAR files. Place your implementation class into that JAR file using the `jar` command.

3. Replace all the *payment_custom.jar* files under the *lib* directory of all the deployed ePayment Manager EAR files with the new *payment_custom.jar*, using jar command.

4. Deploy the new ePayment Manager EAR files on your application server.

5. Go to Payment Settings in the Command Center, and configure the payment gateway(s) to use the new class by replacing the default one, *com.edocs.payment.cassette.ach.AchCheckSubmitPlugIn*, with your new plug-in.

6. Run the pmtCheckSubmit job, which will load the new class from *payment_custom.jar*, because you added it to the classpath of the *MANIFEST* file of *ejb-payment-chksubmit.jar*.

10 Debugging ePayment Manager

Overview

When installing ePayment Manager, follow the installation instructions carefully to set up ePayment Manager correctly. After installation and initial configuration, if you still have problems, the next sections describe a few things you can do to help narrow down the cause.

Viewing WebLogic Logs

From the WebLogic console, you can change the level of log messages. By default, only error messages will be printed out to the console. You can change it to print more detailed information.

View logs from eStatement Manager Command Center

If a ePayment Manager job fails, you can View Logs from the eStatement Manager Command Center to see the details of the error message.

Turning On the ePayment Manager Debug Flag

If you have problems with executing ePayment Manager operations, such as making a check payment or running a payment job, you may want to turn on the *com.edocs.payment.debug* flag to see more details.

Configure your app server so that it uses "-Dcom.edocs.payment.debug=true" as part of the JVM starting option.

For example, for WebLogic on UNIX, change *startWebLogic.sh* to add another option to "java" command:

```
java –Dcom.edocs.payment.debug=true …
```

11 Terminology

Table of Terms

ePayment Manager uses some special terminology, which you should be aware of:

Name	Description
user ID or user login ID	The unique ID that identifies a user in the ePayment Manager system, across different billers. The user must present this user ID in order to login to ePayment Manager system.
payer or payer ID	Same as the user ID.
biller	The entities that issue bills and receive bill payments.
payee or payee ID	The entities that receive bill payments.
payee(ID) reference number	This is a unique number generated internally to identify a payee.
DDN, document definition_name	DDN is a term from eStatement Manager, which defines a bill data stream. A biller in fact can have multiple DDNs. However, currently it is used as the same notion of biller. This may be changed in the future.
user account, user account number	The user's account (number) with the biller.
payer account, payer account number	Same as user account (number).
payment account, payment account number	The user's account (number) with a bank or credit card issuer, used to make payments.
payee account/payee bank account	The payee's account (number) with a bank.

12 Plug-in Sample Code

This chapter shows sample code for the job plug-ins for:

Job	**Plug-in Code**
pmtPaymentReminder	*PaymentReminderPlugIn.java* on page 145
pmtCreditCardSubmit	*VerisignCreditCardSubmitPlugIn.java* on page 148
pmtCheckSubmit	*AchCheckSubmitPlugIn.java* on page 143 *AddendaCheckSubmitPlugIn.java* on page 149 shows an example implementation.
pmtRecurringPayment	*RecurringPaymentPlugIn.javaon* page 146 *SampleRecurringPlugIn.java* on page 151 shows an example implementation.

AchCheckSubmitPlugIn.java

```
package com.edocs.payment.cassette.ach;

import com.edocs.payment.*;
import com.edocs.payment.config.*;
import com.edocs.payment.cassette.CassetteException;
import com.edocs.payment.cassette.CheckSubmitParams;

/**A default implementation for IAchCheckSubmitPlugIn. It does nothing
  *in each method.
  *If you want to write your own implementation, your should derive
  *your implementation from this class and overwrite the
  *methods for which you want to change the behavior.
  */
public class AchCheckSubmitPlugIn implements IAchCheckSubmitPlugIn
{
  private static boolean DEBUG = Boolean.getBoolean("com.edocs.payment.debug");


  public void begin(AchCheckSubmitPlugInParams params) throws CassetteException
  {
    if(DEBUG) System.out.println("In AchCheckSubmitPlugIn.begin()");
  }



  public int preWriteFileHeader(AchCheckSubmitPlugInParams params) throws 
CassetteException
  {
    if(DEBUG) System.out.println("In AchCheckSubmitPlugIn.preWriteFileHeader()");
    return PRE_WRITE_FILE_HEADER_ACCEPT;
  }

  public int preWriteBatchHeader(AchCheckSubmitPlugInParams params) throws 
CassetteException
  {
    if(DEBUG) System.out.println("In AchCheckSubmitPlugIn.preWriteBatchHeader()");
    return 0;
  }

  public int preWriteCheck(AchCheckSubmitPlugInParams params) throws CassetteException
  {
    if(DEBUG) System.out.println("In AchCheckSubmitPlugIn.preWriteCheck(). 
params="+params);
    return PRE_WRITE_CHECK_ACCEPT;
  }

  public int postWriteCheck(AchCheckSubmitPlugInParams params) throws 
CassetteException
  {
    if(DEBUG) System.out.println("In AchCheckSubmitPlugIn.postWriteCheck()");
    return POST_WRITE_CHECK_NOT_MODIFIED;
  }

  public void onWriteCheckException(AchCheckSubmitPlugInParams params)
  {
    if(DEBUG) System.out.println("In AchCheckSubmitPlugIn.onWriteCheckException");
  }

  public int preWriteBatchTrailer(AchCheckSubmitPlugInParams params) throws 
CassetteException
```

```
  {
    if(DEBUG) System.out.println("In AchCheckSubmitPlugIn.preWriteBatchTrailer()");
    return 0;
  }

  public int preWriteFileTrailer(AchCheckSubmitPlugInParams params) throws
CassetteException
  {
    if(DEBUG) System.out.println("In AchCheckSubmitPlugIn.preWriteFileTrailer()");
    return 0;
  }

  public void finish(AchCheckSubmitPlugInParams params) throws CassetteException
  {
    if(DEBUG) System.out.println("In AchCheckSubmitPlugIn.finish()");
  }

  public void abort(AchCheckSubmitPlugInParams params)
  {
    if(DEBUG) System.out.println("In AChCheckSubmitPlugIn.abort()");
  }
}
```

PaymentReminderPlugIn.java

```
package com.edocs.payment.tasks.reminder;

/**This is a default implementation of IPaymentReminderPlugIn. This implementation
  *doesn't doesn nothing in the call back methods. To write your own plug-in,
  *derive your plug-in class from this implementation
  *and overwrite the methods for which you want to change the behavior.
  */
public class PaymentReminderPlugIn implements IPaymentReminderPlugIn
{
  private boolean DEBUG = Boolean.getBoolean("com.edocs.payment.debug");
  public int preSendEmailReminder(PaymentReminderPlugInParams params) throws Exception
  {
      if(DEBUG) System.out.println("PaymentReminderPlugIn.preSendEmailReminder,
reminder="+params.getPaymentReminder());
```

```
      return PRE_SEND_EMAIL_ACCEPT;
  }


  public int preSendEmailCheck(PaymentReminderPlugInParams params) throws Exception
{
      if(DEBUG) System.out.println("PaymentReminderPlugIn.preSendEmailCheck, 
check="+params.getCheck());
        return PRE_SEND_EMAIL_ACCEPT;
}


  public int preSendEmailCreditCard(PaymentReminderPlugInParams params) throws 
Exception
  {
        if(DEBUG) System.out.println("PaymentReminderPlugIn.preSendEmailCreditCard, 
ccard="+params.getCreditCard());
        return PRE_SEND_EMAIL_ACCEPT;
  }


}
```

RecurringPaymentPlugIn.java

```
package com.edocs.payment.tasks.recur_payment;


import com.edocs.payment.*;
import com.edocs.payment.config.*;
import com.edocs.payment.payenroll.*;


/**This class implements IRecurringPaymentPlugIn. It does nothing in each method.
  *When you write your own plug-in, derive your plug-in
  *class from this class, and then overwrite the methods for which you want to
  *change the default behavior.
  */
public class RecurringPaymentPlugIn
      implements IRecurringPaymentPlugIn
{
  private static boolean DEBUG = Boolean.getBoolean("com.edocs.payment.debug");


  public int preGetLatestSummary(SynchronizeRecurringPlugInParams p) throws Exception
  {
    if(DEBUG) System.out.println("RecurringPaymentPlugIn.preGetLatestSummary() is
called");
    return PRE_GET_LATEST_SUMMARY_ACCEPT;


  }


  public int preInsertLatestSummary(SynchronizeRecurringPlugInParams p) throws
Exception
  {
    if(DEBUG) System.out.println("RecurringPaymentPlugIn.preInsertLatestSummary() is
called");
    return PRE_INSERT_LATEST_SUMMARY_ACCEPT;
  }



  public int preUpdateSynchronizedRecurring(SynchronizeRecurringPlugInParams p) throws
Exception
  {
    if(DEBUG)
System.out.println("RecurringPaymentPlugIn.preUpdateSynchronizeRecurring() is
called");
    return PRE_UPDATE_SYNCHRONIZED_RECURRING_ACCEPT;
  }


  public int preSchedulePayment(SchedulePaymentPlugInParams params) throws Exception


  {
    if(DEBUG) System.out.println("SchedulePaymentPlugIn.preSchedulePayment() is
called");
    return PRE_SCHEDULE_PAYMENT_ACCEPT;
  }


  public int preSendEmail(SchedulePaymentPlugInParams params) throws Exception


  {
    if(DEBUG) System.out.println("SchedulePaymentPlugIn.preSendEmail() is called");
    return PRE_SEND_EMAIL_ACCEPT;
  }
}
```

VerisignCreditCardSubmitPlugIn.java

```
package com.edocs.payment.cassette.verisign;


import com.edocs.payment.*;

import com.edocs.payment.config.*;

import com.edocs.payment.cassette.*;


/**This class offers a default implementation for
IVerisignCreditCardSubmitPlugIn.

  *Each method currently does nothing and return directly.

  *You should re-implement this interface if needed.

  *We strongly recommended that you derive your implementation class from
this

  *default implementation.

  */

public class VerisignCreditCardSubmitPlugIn implements
IVerisignCreditCardSubmitPlugIn

{

  private static boolean DEBUG =
Boolean.getBoolean("com.edocs.payment.debug");


  public void begin(VerisignCreditCardSubmitPlugInParams params) throws
CassetteException

  {

    if(DEBUG) System.out.println("In
VerisignCreditCardSubmitPlugIn.begin()");

  }


  public int preAuthorize(VerisignCreditCardSubmitPlugInParams params)

  {

    if(DEBUG) System.out.println("In
VerisignCreditCardSubmitPlugIn.preAuthorize. params="+params);

    return PRE_AUTH_ACCEPT;

  }


  public int postAuthorize(VerisignCreditCardSubmitPlugInParams params)

  {
    if(DEBUG) System.out.println("In 
VerisignCreditCardSubmitPlugIn.postAuthorize");

    return POST_AUTH_NOT_MODIFIED;

  }


  public void onAuthorizeException(VerisignCreditCardSubmitPlugInParams 
params)

  {

    if(DEBUG) System.out.println("In 
VerisignCreditCardSubmitPlugIn.onAuthorizeException");

  }


  public void finish(VerisignCreditCardSubmitPlugInParams params)

  {

    if(DEBUG) System.out.println("In 
VerisignCreditCardSubmitPlugIn.finish()");

  }


  public void abort(VerisignCreditCardSubmitPlugInParams params)

  {

    if(DEBUG) System.out.println("In 
VerisignCreditCardSubmitPlugIn.abort()");

  }

}
```

AddendaCheckSubmitPlugIn.java

```
package com.edocs.payment.cassette.ach;


import com.edocs.payment.*;

import com.edocs.payment.config.*;

import com.edocs.payment.db.*;

import com.edocs.payment.cassette.CassetteException;

import java.util.*;


/**This plug-in demonstrates how to append a list of addenda records to

  *a check payment record in an ACH file. Addenda information is biller-specific.

  *You should write your own implementation to retrieve the addenda information
  *for a particular biller.
  */
public class AddendaCheckSubmitPlugIn extends AchCheckSubmitPlugIn implements
IAchCheckSubmitPlugIn
{
  private static boolean DEBUG = Boolean.getBoolean("com.edocs.payment.debug");

  /**This method calls Addenda.setAddendaNote() to set the addenda information
    *of a check payment. The addenda information actually comes from the
    *invoices of the check payment. This method first checks whether there are
    *invoices associated with this check. If so, it retrieves the invoices, and
    *for each invoice creates an Addenda record whose addenda note is set
    *to a format like "invoiceNumber=..., invoiceAmount=...".
    *@param params An AchCheckSubmitPlugInParams object.
    *@return IAchCheckSubmitPlugIn.PRE_WRITE_CHECK_ACCEPT
    */
  public int preWriteCheck(AchCheckSubmitPlugInParams params)
  {
    if(params.isPrenote())
      return PRE_WRITE_CHECK_ACCEPT;

    Invoice invoice;
    List invoices = null;

    if(DEBUG) System.out.println("In AchCheckSubmitPlugIn.preWriteCheck(),
check="+params.getCheck());
    // retrieve invoice info, put into params.
    //

    PaymentQueryParams query_param = new PaymentQueryParams();

    IPaymentInvoiceLog pilog = PaymentDBFactory.newPaymentInvoiceLog();

    query_param.setPaymentId(params.getCheck().getPaymentId());

    try {
        invoices = pilog.query(query_param);
    } catch (Throwable e) { }
    Iterator iter = invoices.iterator();

    List addendas = new LinkedList();

    while ( iter.hasNext() ) {
           invoice = (Invoice)iter.next();
           Addenda addenda = new Addenda();

addenda.setAddendaNote("invoiceNumber="+invoice.getInvoiceNumber()+",invoiceAmount="+i
nvoice.getInvoiceAmount());



           addendas.add(addenda);
    }
    params.setAddendas(addendas);
    return PRE_WRITE_CHECK_ACCEPT;
  }



}
```

SampleRecurringPlugIn.java

```
package com.edocs.payment.tasks.recur_payment;

import java.util.*;
import com.edocs.payment.*;
import com.edocs.payment.config.*;
import com.edocs.payment.payenroll.*;
import com.edocs.payment.util.template.*;

/**This sample recurring payment plug-in demonstrates how to fill in the
  *flexible fields of IPaymentTransaction (check or credit card) with the
  *information retrieved from IBillSummary.
  */
public class SampleRecurringPlugIn
       extends RecurringPaymentPlugIn implements IRecurringPaymentPlugIn
{
  private static boolean DEBUG = Boolean.getBoolean("com.edocs.payment.debug");
  /**Must have this default constructor.
    */
  public SampleRecurringPlugIn()
  {
  }

  /**This method is called before the pmtRecurPayment job tries to get the latest bill
summary
    *for a user account. This implementation is empty (does nothing).
    *@param p  A SynchronizeRecurringPlugInParams object.
    *@return IRecurringPaymentPlugIn.PRE_GET_LATEST_SUMMARY_ACCEPT
    */
  public int preGetLatestSummary(SynchronizeRecurringPlugInParams p) throws Exception
  {
    if(DEBUG) System.out.println("RecurringPaymentPlugIn.preGetLatestSummary() is
called");
    if(p.getPaymentConfig() == null)
      throw new Exception("config is not set");

    return PRE_GET_LATEST_SUMMARY_ACCEPT;

  }

  /**This method is called before the pmtRecurPayment job inserts the latest summary
   *into the Payment table. The IBillSummary object has a list of extended attributes
   *which can hold any bill summary information not required by Payment.
   *However, these extended attributes won't be inserted into
   *Payment database. This method checks whether there are at least two extended
attributes
   *in the summary, and if so, fills the two flexible fields, 1 and 2, of IBillSummary
   *with the first and second extended attributes, respectively. The two flexible
   *fields are inserted into the Payment database by the pmtRecurPayment job.
   *@param p A SynchronizeRecurringPlugInParams object.
   *@return int; IRecurringPaymentPlugIn.PRE_INSERT_LATEST_SUMMARY_ACCEPT
   */
  public int preInsertLatestSummary(SynchronizeRecurringPlugInParams p) throws
Exception
  {
    if(DEBUG) System.out.println("RecurringPaymentPlugIn.preInsertLatestSummary() is 
called");
    IBillSummary sum = p.getBillSummary();
    if(sum != null)
    {
      Map attrs = sum.getExtendedAttributes();
      if(attrs != null && attrs.size() >= 2){
        Object[] keys = attrs.keySet().toArray();
        sum.setFlexibleField1((String)attrs.get(keys[0]));
        sum.setFlexibleField2((String)attrs.get(keys[1]));
        if(DEBUG) System.out.println("RecurringPaymentPlugIn, summary flex fields set. 
sum="+sum);
      }
    }
    return PRE_INSERT_LATEST_SUMMARY_ACCEPT;
  }



  /**This method is called before the pmtRecurPayment job writes the "synchronized"
   * recurring payment back to the database. A "synchronized" recurring payment
   * means that there is a new bill that needs to be paid. This method fills
   * the flexible fields 1 and 2 of current IRecurringPayment with the
   * flexible fields 1 and 2 of current IBillSummary, respectively. The recurring
   * job then updates the IRecurringPayment into the database.
   *@param p A SynchronizeRecurringPlugInParams object.
   *@return int; IRecurringPaymentSummary.PRE_UPDATE_SYNCHRONIZED_RECURRING_ACCEPT
   */
  public int preUpdateSynchronizedRecurring(SynchronizeRecurringPlugInParams p) throws 
Exception
  {
    if(DEBUG) 
System.out.println("RecurringPaymentPlugIn.preUpdateSynchronizeRecurring() is 
called");
    IBillSummary sum = p.getBillSummary();
    IRecurringPayment rec = p.getRecurringPayment();
    if(sum != null && rec != null)
    {
      Map attrs = sum.getExtendedAttributes();
      if(attrs != null && attrs.size() >= 2){
        Object[] keys = attrs.keySet().toArray();
        rec.setFlexibleField1(sum.getFlexibleField1());
        rec.setFlexibleField2(sum.getFlexibleField2());
        if(DEBUG) System.out.println("RecurringPaymentPlugIn, recurring flex fields
set. rec="+rec);
      }
    }
    return PRE_UPDATE_SYNCHRONIZED_RECURRING_ACCEPT;
  }

  /**This method is called before the pmtRecurPayment job schedules (inserts) a new
    *payment into the Payment database. This method fills in the flexible
    *fields 1 and 2 of the payment( check or credit card) with the flexible
    *fields 1 and 2 of the IRecurringPayment, respectively. The job then
    *inserts the payment with the flexible fields into database.
    *@param params A SchedulePaymentPlugInParams object.
    *@return IRecurringPayment.PRE_SCHEDULE_PAYMENT_ACCEPT
    */
  public int preSchedulePayment(SchedulePaymentPlugInParams params) throws Exception

  {
    if(DEBUG) System.out.println("SchedulePaymentPlugIn.preSchedulePayment() is
called");
    IPaymentTransaction tran = params.getPayment();
    IRecurringPayment rec  = params.getRecurringPayment();
    if(rec != null && tran != null){
      if(tran instanceof ICheck){
        ((ICheck)tran).setFlexibleField1(rec.getFlexibleField1());
        ((ICheck)tran).setFlexibleField2(rec.getFlexibleField2());
      }else{
        ((ICreditCard)tran).setFlexibleField1(rec.getFlexibleField1());
        ((ICreditCard)tran).setFlexibleField2(rec.getFlexibleField2());
      }

    }
    return PRE_SCHEDULE_PAYMENT_ACCEPT;
  }

  /**This method is called before the pmtRecurPayment job sends an email to the user.
```

```
   * The passed in SchedulePaymentPlugInParams parameter includes the mail-to address
and subject.
   * You can use this method to check/change the mail-to addresses and subject.
   * The mail-to addresses and subject of SchedulePaymentPlugInParams
   * will be passed back to Payment and used by Payment to send out email.
   *@param params A SchedulePaymentPlugInParams object.
   *@return IRecurringPayment.PRE_SEND_EMAIL_ACCEPT
   */
  public int preSendEmail(SchedulePaymentPlugInParams params) throws Exception
  {
    if(DEBUG) System.out.println("SchedulePaymentPlugIn.preSendEmail() is called");
    params.setMailSubject("Hi, this subject is set by SampleRecurringPaymentPlugIn");
    return PRE_SEND_EMAIL_ACCEPT;
  }
}
```

13 Auditing

ePayment Manager audits some payment jobs to track a variety of transaction failures. Audits are kept for actions taken through the UI, as well as jobs.

Jobs Audited

The jobs that write to the audit tables are listed below along with the information that is audited.

pmtCheckSubmit job

- Payments that failed during submission
- Encryption exceptions

pmtPaymentReminder

Payment reminders that were not sent, including:

- Regular payment reminders that failed to send, for any reason, such as bad email address.
- Check payment emails that failed to send, for any reason, such as encryption error, bad email address.
- Credit card payment emails failed to send, for any reason, such as encryption error or bad email address.

pmtCreditCardSubmit

Credit card payments failed to submit, for example, because of encryption errors, invalid credit card information (such as invalid account) or network errors.

pmtIntegrator (AR) job

Check and credit card payments that were not written to the AR file. For example, because of encryption errors or file write errors.

pmtRecurringPayment

Check and credit card payments that failed.

pmtCheckSubmit and pmtCreditCardSubmit

UI Actions Audited

This audit lists successful and unsuccessful payments along with a reason code.

The UI actions that trigger an audit entry are:

- Create Recurring Payment
- Update Recurring Payment
- Delete Recurring Payment
- Create Schedule Payment
- Create Instant Payment
- Cancel Future Payment - Credit Card Payment
- Update Future Payment - Credit Card Payment
- Cancel Future Payment - Check Payment
- Update Future Payment - Check Payment
- Create Payment Reminder
- Update Payment Reminder
- Delete Payment Reminder
- Create Check Account
- Edit Check Account
- Delete Check Account
- Create Credit Card Account
- Edit Credit Card Account
- Delete Credit Card Account
- Initiating a Payment Refund in the UI
- Refund a payment using the Payment History page
- View an invoice using the Payment History page
- Create a new external payment on the External Payments page
- Update an existing payment using the External Payments History page
- Cancel a payment using the External Payments History page

Example of UI Audit Flow

1. The customer selects the **Setup of recurring payment** option, populates the information to initially set up recurring payment, and submits it. The following information is recorded as the audit data in the *recurring_payments_history* table in addition to the columns defined in the *recurring _payments* table. (This history table contains all the columns defined in the recurring_payments (regular table) table plus the additional following columns).

Column	**Value**	**Description**
audit_operation	1001	This constant value for the operation is explained in the recurring_payment_const table.
audit_status	1	Status constant value successful operation. This constant value for the status is explained in the *recurring_payment_const* table.
audit_reason		Description of the audit.
Job_id	0	Since this is an UI operation, job_id value is 0 (not a job).
Job_name	NULL	Since this is a UI operation, job_name is NULL.
Timestamp		The current system time when an audit is taken place.

2. The customer selects **Recurring Payment** option, selects Update, and updates the recurring payment information and submits it. The following information is recorded as the audit data in recurring_payments_history table other than the columns defined in the regular recurring _payments table. (This history table contains all the columns defined in the recurring_payments (regular table) table and additional following columns).

Column	**Value**	**Description**
audit_operation	1002	This constant value for the operation is explained in the *recurring_payment_const* table.
audit_status	1	Status constant value for successful operation. This constant value for the status is explained in the *recurring_payment_const* table.
audit_reason		Description of the audit.
Job_id	0	Since this is a UI operation, job_id value is 0.
Job_name	NULL	Since this is a UI operation, job_name is NULL.
Timestamp		The current system time when an audit is taken place.

3. The customer selects **Recurring Payment** option, and then selects Delete, the following information is recorded as the audit data in recurring_payments_history table other than the columns defined in the regular recurring _payments table. (This history table contains all the columns defined in the recurring_payments (regular table) table and additional following columns).

Column	Value	Description
audit_operation	1003	This constant value for the operation is explained in the *recurring_payment_const* table.
audit_status	1	Status constant value for successful operation. This constant value for the status is explained in the *recurring_payment_const* table.
audit_reason		Description of the audit.
Job_id	0	Since this is a UI operation, job_id value is 0.
Job_name	NULL	Since this is a UI operation, job_name is NULL.
Timestamp		The current system time when an audit is taken place.

4. The customer selects **Create Check account** in the "User Profile" UI, and submits the new check account information, the following audit data is recorded in payment_accounts_history table other than the columns defined in the regular payment_accounts table. (This history table contains all the columns defined in the payment_accounts (regular table) table and additional following columns).

Column	Value	Description
audit_operation	1001	This constant value for the operation is explained in the *payment_account_const* table.
audit_status	1	Status constant value for successful operation. This constant value for the status is explained in the *payment_account_const* table.
audit_reason		Description of the audit.
Job_id	0	Since this is a UI operation, job_id value is 0.
Job_name	NULL	Since this is a UI operation, job_name is NULL.
Timestamp		The current system time when an audit is taken place.

5. The customer selects **Update Check account** in the "User Profile" UI, and submits the updated check account information, the following audit data is recorded in payment_accounts_history table other than the columns defined in the regular payment_accounts table. (This history table contains all the columns defined in the payment_accounts (regular table) table and additional following columns).

Column	Value	Description
audit_operation	1002	This constant value for the operation is explained in the *payment_account_const* table.
audit_status	1	Status constant value for successful operation. This constant value for the status is explained in the *payment_account_const* table.
audit_reason		Description of the audit.

Column	Value	Description
Job_id	0	Since this is a UI operation, job_id value is 0.
Job_name	NULL	Since this is a UI operation, job_name is NULL.
Timestamp		The current system time when an audit is taken place.

6. The customer selects **Delete Check account** in the "User Profile" UI, and submits the delete request, the following audit data is recorded in payment_accounts_history table other than the columns defined in the regular payment_accounts table. (This history table contains all the columns defined in the payment_accounts (regular table) table and additional following columns).

Column	Value	Description
audit_operation	1003	This constant value for the operation is explained in the payment_account_const table).
audit_status	1	Status constant value for successful operation. (This constant value for the status is explained in the payment_account_const table.)
audit_reason		Description of the audit.
Job_id	0	Since this is a UI operation, job_id value is 0.
Job_name	NULL	Since this is a UI operation, job_name is NULL.
Timestamp		The current system time when an audit is taken place.

7. The customer selects **Create Credit Card account** in the "User Profile" UI, and submits the new credit card account information, the following audit data is recorded in payment_accounts_history table other than the columns defined in the regular payment_accounts table. (This history table contains all the columns defined in the payment_accounts (regular table) table and additional following columns).

Column	Value	Description
audit_operation	1001	This constant value for the operation is explained in the *payment_account_const* table.
audit_status	1	Status constant value for successful operation. This constant value for the status is explained in the *payment_account_const* table.
audit_reason		Description of the audit.
Job_id	0	Since this is a UI operation, job_id value is 0.
Job_name	NULL	Since this is a UI operation, job_name is NULL.
Timestamp		The current system time when an audit is taken place.

8. The customer selects **Update Credit Card account** in the “User Profile” UI, and submits the updated credit card account information, the following audit data is recorded in payment_accounts_history table other than the columns defined in the regular payment_accounts table. (This history table contains all the columns defined in the payment_accounts (regular table) table and additional following columns).

Column	**Value**	**Description**
audit_operation	1002	This constant value for the operation is explained in the *payment_account_const* table.
audit_status	1	Status constant value for successful operation. This constant value for the status is explained in the *payment_account_const* table.
audit_reason		Description of the audit.
Job_id	0	Since this is a UI operation, job_id value is 0.
Job_name	NULL	Since this is a UI operation, job_name is NULL.
Timestamp		The current system time when an audit is taken place.

9. The customer selects **Delete Credit Card account** in the “User Profile” UI, and submits the delete request, the following audit data is recorded in payment_accounts_history table other than the columns defined in the regular payment_accounts table. (This history table contains all the columns defined in the payment_accounts (regular table) table and additional following columns).

Column	**Value**	**Description**
audit_operation	1003	This constant value for the operation is explained in the *payment_account_const* table.
audit_status	1	Status constant value for successful operation. This constant value for the status is explained in the *payment_account_const* table.
audit_reason		Description of the audit.
Job_id	0	Since this is a UI operation, job_id value will be 0.
Job_name	NULL	Since this is a UI operation, job_name will be NULL.
Timestamp		The current system time when an audit is taken place.

10. The customer selects **Create payment reminder** in the “User Profile” UI, and submits the new payment reminder information, the following audit data is recorded in payment_reminders_history table other than the columns defined in the regular payment_reminders table. (This history table contains all the columns defined in the payment_reminders (regular table) table and additional following columns).

Column	**Value**	**Description**
audit_operation	1001	This constant value for the operation is explained in the *payment_reminder_const* table.
audit_status	1	Status constant value for successful operation. This constant value for the status is explained in the *payment_reminder_const* table.
audit_reason		Description of the audit.
Job_id	0	Since this is a UI operation, job_id value is 0.
Job_name	NULL	Since this is a UI operation, job_name is NULL.
Timestamp		The current system time when an audit is taken place.

11. The customer selects **Update payment reminder** in the "User Profile" UI, and submits the updated payment reminder information, the following audit data is recorded in payment_reminders_history table other than the columns defined in the regular payment_reminders table. (This history table contains all the columns defined in the payment_reminders (regular table) table and additional following columns).

Column	**Value**	**Description**
audit_operation	1002	This constant value for the operation is explained in the *payment_reminder_const* table.
audit_status	1	Status constant value for successful operation. This constant value for the status is explained in the *payment_reminder_const* table.
audit_reason		Description of the audit.
Job_id	0	Since this is a UI operation, job_id value is 0.
Job_name	NULL	Since this is a UI operation, job_name will be NULL.
Timestamp		The current system time when an audit is taken place.

12. The customer selects **Delete payment reminder** in the "User Profile" UI, and submits the delete request for the payment reminder, the following audit data is recorded in payment_reminders_history table other than the columns defined in the regular payment_reminders table. (This history table contains all the columns defined in the payment_reminders (regular table) table and additional following columns).

Column	**Value**	**Description**
audit_operation	1003	This constant value for the operation is explained in the *payment_reminder_const* table.

Column	**Value**	**Description**
audit_status	1	Status constant value for successful operation. This constant value for the status is explained in the *payment_reminder_const* table.
audit_reason		Description of the audit.
Job_id	0	Since this is a UI operation, job_id value is 0.
Job_name	NULL	Since this is a UI operation, job_name will be NULL.
Timestamp		The current system time when an audit is taken place.

Query Files

The following files are provided for each platform to support queries of the audit tables.

SQL2000

Windows

getAuditDataByPid.bat

getAuditDataByPaymentId.bat

getAuditDataByAccount.bat

ORACLE

Windows

getAuditDataByAccount.bat

getAuditDataByPaymentId.bat

getAuditDataByPid.bat

set_audit_isql_options.bat

getAuditDataByAccount.sql

getAuditDataByPaymentId.sql

getAuditDataByPid.sql

getAuditInfoByAccount.sql

getAuditInfoByPaymentId.sql

getAuditInfoByPid.sql

UNIX

getAuditInfoByAccount.sh

getAuditInfoByPaymentId.sh

getAuditInfoByPid.sh

getAuditDataByAccount.sql

getAuditDataByPaymentId.sql

getAuditDataByPid.sql

getAuditInfoByAccount.sql

getAuditInfoByPaymentId.sql

getAuditInfoByPid.sql

DB2

Windows

getAuditDataByAccount.bat

getAuditDataByPaymentId.bat

getAuditDataByPid.bat

set_audit_isql_options.bat

UNIX

getAuditDataByAccount.sh

getAuditDataByPaymentId.sh

getAuditDataByPid.sh

getAuditInfoByAccount.sh

getAuditInfoByPaymentId.sh

getAuditInfoByPid.sh

Running Audit Queries

Audit queries require on of the following arguments:

- Payment ID
- User Account Number
- PID

The audit queries are implemented in batch files, which require the user argument and date range. The results are displayed on the console.

Before running the queries, you must preform setup. The description for each query describes the setup.

Query Audit data by Payment ID

Displays data from all history tables which have a payment ID column. This query performs a simple select on each table where the Payment ID matches and the time_stamp is between "fromTime" and "toTime". The following tables are queried:

- check_payments_history
- creditcard_payments_history
- payment_bill_summaries_history
- payment_email_history

Query Audit data by User Account Number

Displays data from all history tables which have a payer ID column. This query performs a simple select on each table where the payer ID matches "Account Number", and whose time_stamp is between "fromTime" and "toTime". The AccountNumber is the account number with the biller (*payee_id* column). The following tables are queried:

- check_payments_history
- creditcard_payments_history
- payment_bill_summaries_history
- recurring_payments_history

Query Audit data by PID

Displays data from all the history tables which have a PID column. This query performs a simple select on each table where the PID matches and whose time_stamp is between "fromTime" and "toTime". The following tables are queried:

- check_payments_history
- creditcard_payments_history
- payment_accounts_history
- recurring_payments_history

Query Setup

Before running the queries, you must:

1. Set the database connection parameters
2. Configure TNS Listener for Oracle (Client/Server)
3. Configure DB2 Clients for windows platform
4. Check execution permissions for shell scripts

5. Database connection parameters

Configuration for each platform is described below:

Windows Configuration

For Windows *set_isql_options.bat* must be edited before running the queries. The file constrains the following line:

```
set ISQL_OPTIONS=-U <username> -P <password> -S <sqlsvr-
Servername> -d <database name>
```

Edit this file and enter your values for username, password, server name and database name. For example:

```
set ISQL_OPTIONS=-U edx1 -P edx1 -S EDXSERVER -d edxDB
```

UNIX Configuration

For UNIX platforms, the database connection string is embedded in the file. You must edit the connection parameters in each file before running the queries. The connection parameters are as follows:

For DB2:

```
db2 connect to <database> user <username>  using <password>
```

For example:

```
db2 connect to EDXDB41L user db2inst1  using db2admin
```

For Oracle:

```
sqlplus <username>/<password>@<TNS name>
```

For example:

```
sqlplus edx1/edxadmin@edxdb
```

TNS Listener for Oracle (Client/Server)

The TNS Listener has to be configured for Oracle DB in Windows and UNIX platforms for client / server.

Permissions on UNIX platform

Execution permissions for shell scripts should be granted to run the shell scripts successfully. For example:

```
$ chmod 755 *.sh
```

Running the Queries in Windows

MSSQL

Q1: Query Audit data by Payment ID

Change your working directory to the location of the query script files, and run *getAuditDataByPaymentId.bat*. This file requires three parameters: Payment ID, From Timestamp, and To Timestamp. The execution format is:

```
getAuditDataByPaymentId <Payment ID>,<from date>,<to date>
```

For example:

```
getAuditDataByPaymentId 123465564,'2003-01-01’,'2004-12-12'
```

Where:

Date format is YYYY-MM-DD

Payment ID is numeric

Q2: Query Audit data by Account

Change your working directory to the location of the query script files, and run *getAuditDataByAccount.bat*. This file requires three parameters: Account Number, From Timestamp, and To Timestamp. The execution format is:

```
getAuditDataByAccount <account_number>,<from date>,<to date>
```

For example:

```
getAuditDataByAccount '123465564','2003-01-01’,'2004-12-12'
```

Where:

Date format is YYYY-MM-DD

Account Number is a string

Q3: Query Audit data by PID

Change your working directory to the location of the query script files, and run getAuditDataByPid.bat. This file requires three parameters: PID, From Timestamp, and To Timestamp. The execution format is:

```
getAuditDataByPid <pid>,<from date>,<to date>
```

For example:

```
getAuditDataByPid '123465564','2003-01-01’,'2004-12-12'
```

Where:

Date format is YYYY-MM-DD

PID is a string

Oracle

Q1: Query Audit data by Payment ID

Change your working directory to the location of the query script files, and run *getAuditDataByPaymentId.bat*. This file requires three parameters: Payment ID, From Timestamp, and To Timestamp. The execution format is:

```
getAuditDataByPaymentId <Payment ID>,<from date>,<to date>
```

For example:

```
getAuditDataByPaymentId 123465564,'2003-01-01’,'2004-12-12'
```

Where:

Date format is YYYY-MM-DD.

Payment ID is numeric

Q2: Query Audit data by Account

Change your working directory to the location of the query script files, and run *getAuditDataByAccount.bat*. This file requires three parameters: Account Number, From Timestamp, and To Timestamp. The execution format is:

```
getAuditDataByAccount <account_number>,<from date>,<to date>
```

For example:

```
getAuditDataByAccount '123465564','2003-01-01’,'2004-12-12'
```

Where:

Date format is YYYY-MM-DD

Account Number is a string

Q3: Query Audit data by PID

Change your working directory to the location of the query script files, and run “getAuditDataByPid.bat”. This file requires three parameters: PID, From Timestamp, and To Timestamp. The execution format is:

```
getAuditDataByPid <pid>,<from date>,<to date>
```

For example:

```
getAuditDataByPid '123465564','2003-01-01’,'2004-12-12'
```

Where:

Date format is YYYY-MM-DD

PID is a string

DB2

Q1: Query Audit data by Payment ID

Change your working directory to the location of the query script files, and run *getAuditDataByPaymentId.bat*. This file requires three parameters: Payment ID, From Timestamp, and To Timestamp. The execution format is:

```
getAuditDataByPaymentId <Payment ID>,<from date>,<to date>
```

For example:

```
getAuditDataByPaymentId 123465564,'2003-01-01’,'2004-12-12'
```

Where:

Date format is YYYY-MM-DD

Payment ID is not a string it is a numeric value

Q2: Query Audit data by Account

Change your working directory to the location of the query script files, and run *getAuditDataByAccount.bat*. This file requires three parameters: Account Number, From Timestamp, and To Timestamp. The execution format is:

```
getAuditDataByAccount <account_number>,<from date>,<to date>
```

For example:

```
getAuditDataByAccount '123465564','2003-01-01’,'2004-12-12'
```

Where:

Date format is YYYY-MM-DD.

Account Number is a string.

Q3: Query Audit data by PID

Change your working directory to the location of the query script files, and run getAuditDataByPid.bat. This file requires three parameters: PID, From Timestamp, and To Timestamp. The execution format is:

```
getAuditDataByPid <pid>,<from date>,<to date>
```

For example:

```
getAuditDataByPid '123465564','2003-01-01’,'2004-12-12'
```

Where:

Date format is YYYY-MM-DD

PID is a string

Running the Queries in UNIX

Oracle

Q1: Query Audit data by Payment ID

Change your working directory to the location of the query script files, and run *getAuditInfoByPaymentId.sh*. This file requires three parameters: Payment ID, From Timestamp, and To Timestamp. The execution format is:

```
$ ./getAuditInfoByPaymentId.sh <Payment ID> <from date> <to
date>
```

For example:

```
$ ./getAuditInfoByPaymentId.sh 123465564 '2003-01-01’ '2004-
12-12'
```

Where:

Date format is YYYY-MM-DD

Payment ID is numeric

Arguments are separated by spaces

Q2: Query Audit data by Account

Change your working directory to the location of the query script files, and run *getAuditInfoByAccount.sh*. This file requires three parameters: Account Number, From Timestamp, and To Timestamp. The execution format is:

```
$ ./getAuditInfoByAccount.sh <account_number> <from date> <to
date>
```

For example:

```
& ./getAuditInfoByAccount.sh '123465564' '2003-01-01’ '2004-
12-12'
```

Where:

Date format is YYYY-MM-DD

Account Number is a string

Arguments are separated by spaces

Q3: Query Audit data by PID

Change your working directory to the location of the query script files, and run *getAuditInfoByPid.sh*. This file requires three parameters: PID, From Timestamp, and To Timestamp. The execution format is:

```
$ ./getAuditInfoByPid.sh <pid> <from date> <to date>
```

For example:

```
$ ./getAuditInfoByPid '123465564' '2003-01-01’ '2004-12-12'
```

Where:

Date format is YYYY-MM-DD

PID is a string

Arguments are separated by spaces

DB2

Q1: Query Audit data by Payment ID

Change your working directory to the location of the query script files, and run *getAuditDataByPaymentId.sh*. This file requires three parameters: Payment ID, From Timestamp, and To Timestamp. The execution format is:

```
$ ./getAuditDataByPaymentId.sh <Payment ID> <from date> <to
date>
```

For example:

```
$ ./getAuditDataByPaymentId.sh 123465564 '2003-01-01’ '2004-
12-12'
```

Where:

Date format is YYYY-MM-DD

Payment ID is not a string it is a numeric value

Arguments are separated by spaces

Q2: Query Audit data by Account

Change your working directory to the location of the query script files, and run “getAuditDataByAccount.sh”. This file requires three parameters: Account Number, From Timestamp, and To Timestamp. The execution format is:

```
$ ./getAuditDataByAccount.sh <account_number> <from date> <to
date>
```

For example:

```
$ ./getAuditDataByAccount.sh '123465564' '2003-01-01’ '2004-
12-12'
```

Where:

Date format is YYYY-MM-DD

The Account Number is a string

Arguments are separated by spaces

Q3: Query Audit data by PID

Change your working directory to the location of the query script files, and run *getAuditDataByPid.sh*. This file requires three parameters: PID, From Timestamp, and To Timestamp. The execution format is:

```
$ ./getAuditDataByPid.sh <pid> <from date> <to date>
```

For example:

```
$ ./getAuditDataByPid.sh '123465564' '2003-01-01’ '2004-12-12'
```

Where:

Date format is YYYY-MM-DD

PID is a string

Arguments are separated by spaces

Audit Database

The eStatement Manager/Payment database has been updated to support auditing.

Modified Tables

The following tables have the new columns:

- check_payments_history
- creditcard_payments_history

The history tables have all the columns that the base table has (*check_payments* and *creditcard_payments*), plus the following columns:

Column Name	Comments
audit_operation	Defined in corresponding constant tables
audit_status	Defined in corresponding constant tables
audit_reason	Description of the audit
job_id	Pwc job ID
job_name	User given job name (see Job Name Entries)
time_stamp	The record insertion time. For example: 1/18/2004 11:47:38 AM

New Tables

All the following tables are based on the table name with "_history" at the end. They have all the columns in the base table, plus the new columns listed in the preceding table (in the *Modified Tables* section) to support audit.

- payment_accounts_history
- payment_bill_summeries_history
- payment_reminder_history
- recurring_payments_history

payment_email_history

This table is new, and not based on a previous table. It has the following columns, plus the columns listed in the preceding table (in the *Modified Tables* section) to support audit.

Column Name	Comments
type	This indicates the purpose of the email. Possible values are listed in the table ‘Email Types’ below.
payee id	DDN
payer_id	User ID
account_numer	Check or credit card number
payment_id	Payment ID
to_address	Receivers email address. If there are multiple addresses, they will be in semicolon separated.
content	Note: Actual length of the email content must be truncated based on job configuration, “Email Content Audit Length”.
audit_operation	Defined in corresponding constant tables
audit_status	Defined in corresponding constant tables
audit_reason	Description of the audit
job_id	Pwc job ID
job_name	User given job name (see Job Name Entries)
time_stamp	The record insertion time. For example: 1/18/2004 11:47:38 AM

The following table lists the possible values for email types and description.

Email Type	Description
0	Unknown email type.
1	A fixed date payment reminder email.
2	Before due date payment reminder email.
3	After due date payment reminder email.
4	Check status notification email.
5	Credit card status notification email.
6	Recurring payment cancelled email.

Email Type	**Description**
7	Recurring payment scheduled email.
8	Payment account status notification email.
9	Credit card expiration notification email.

Audit Table Constants

The following table lists the tables that have audit information, and the names of the corresponding code tables that explain the numeric codes for audit columns. See the tables in your Payment database for the latest descriptions for each code.

Constant Table name	**History table name**
credit_card_const	creditcard_payments_history
check_const	check_payments_history
recurring_payment_const	recurring_payment_history
payment_email_const	payment_email_history
payment_bill_summaries_const	payment_bill_summaries history
payment_account_const	payment_accounts_history
payment_reminders_const	payment_reminders_history

Job Name Entries

User job names are combined with a shortened version of the task name to keep database entries manageable. The name of the job given by the user is combined with a shortened version of the task name as follows:

```
<job name given by the Admin>-<shorten task name>
```

The following table shows the shortened name for each job.

Task name	**Shortened task name**
CheckSubmitTask	ChkSubTsk
CheckUpdateTask	ChkUpdTsk
PaymentIntegratorTask	PmtIntTsk
CreditCardExpNotifyTask	CCExpNTsk
CreditCardSubmitTask	CCSubTsk
CreditCardUpdateTask	CCUpdTsk
ConfirmEnrollTask	ConEnrTsk

Task name	**Shortened task name**
NotifyEnrollTask	NotEnrTsk
RecurPaymentSchedulerTask	RcuSchTsk
RecurPaymentSynchronizerTask	RcuSynTsk
PaymentReminderTask	PmtRmdTsk
SubmitEnrollTask	SubEnrTsk
CustomTask	CustomTsk

14 Implementing a Custom ePayment Manager Cartridge

Demonstration Cartridge

Payment provides an example cartridge that demonstrates how to implement a custom cartridge. The code is in */vobs/payment/com/edocs/payment/cassette/demo*. There are two cartridges:

demo_CheckCassette.java for check payments

demo_CreditCardCassette.java for credit card payments

The example cartridge delegates all API calls to *demo_CheckProcessorProxy.java* and *demo_CreditCardProcessorProxy.java* to communicate with a dummy payment gateway.

If you configure a DDN to use the demonstration cartridge, then you can make payments against it from the user interface.

Implementing Custom Credit Card Cartridge

The example cartridge is based on the interface *com.edocs.payment.cassette.ICreditCardCassette*, which extends from *com.edocs.payment.cassette.IPaymentCassette*, which then extends from *com.edocs.payment.cassette.IEnrollmentCassette*. In general, you don't need to modify `IEnrollmentCassette`, since it defines how to verify a credit card when a user enrolls it through the user interface.

To implement the cartridge, extend your cartridge implementation from `PaymentCassette`, and implement `ICreditCardCassette`.

```
public class MyCreditCardCassette extends PaymentCassette
implements ICreditCardCassette
```

Use *demo_CreditCardCassette.java* to create your implementation. The three methods you should consider implementing are:

```
IPaymentCassette.getDefaultConfigAttributes()

ICreditCardCassette.authorize()

ICreditCardCassette.batchAuthorize()
```

You must implement `IPaymentCassette.getDefaultConfigAttributes()` to return a list of parameters (of type *com.edocs.payment.config.Attribute*), which are used to configure the cartridge. Calling `IPaymentCassette.getDefaultConfigAttributes()` causes those parameters to be displayed in the Payment Settings of the Command Center, where you can use them to configure the cartridge. These parameters include the global ones, the ones shared by both credit card and check types, and the ones specific to this credit card cartridge. Your implementation of `getDefaultConfigAttributes()` must at least return the global and shared parameters in that list. See `demo_CreditCardCassette.getDefaultConfigAttributes()` in the Payment JavaDoc, and the file *demo_CreditCardAttributes.java* for more information.

If you wish to support instant payments, then you must implement the `ICreditCardCassette.authorize()` method. In this method, you must get the payment information from the `ICreditCard` object that is passed in, then send it to the payment gateway. The payment gateway will send back a response, which you will use to update the status of the `ICreditCard` object, as described below:

1. If the payment is authorized, set the status to "settled" by calling:

```
ICreditCard.setStatus(CreditCardState.SETTLED);
```

2. If the payment failed authorization, set status to "failed-authorize" by calling:

```
ICreditCard.setStatus(CreditCardState.FAILED_AUTHORIZE);
```

   You may also want to call `ICreditCard.setTxnErrMsg()` to log an error message.

3. If there is a system or network error (Payment failed to connect to payment gateway), set the status to "failed" by calling:

```
ICreditCard.setStatus(CreditCardState.FAILED);
```

   You may also want to call `ICreditCard.setTxnErrMsg()` to log an error message.

   When you call these methods, Payment updates the credit card information in the database. The ePayment Manager JSP pages get the credit card information from the user and pass the information to the cartridge. After the card is processed, ePayment Manager updates ePayment Manager database.

   If your application will support scheduled payments, then you must implement `ICreditCardCassette.batchAuthorize()`. This method is called by the CreditCardSubmit job, which extracts all the scheduled payments from the database and sends them to the payment gateway. Your cartridge should do the following things:

4. Get the scheduled payments from ePayment Manager database. There are examples of using the APIs in `demo_CreditCardCassette.batchSubmit()`.

5. Loop through the list of payments and send them to the payment gateway. The status of each payment should be set the same way as for instant payments. After setting the status and other information, call the ePayment Manager API to update this credit card back to ePayment Manager database (note that this is different from Instant payments, because ePayment Manager does not update the database).

6. Package your custom cartridge.

   If you are using ePayment Manager 2.2 with WebSphere, you should package it into *payment_client.jar* which is in the *lib* dir of each ePayment Manager EAR.

   If you are using ePayment Manager 3.0 with WebLogic, you should package it into *payment_custom.jar* which is in the *lib* directory of each ePayment Manager EAR.

7. Pre-populate ePayment Manager database.

   Tell ePayment Manager about your cartridge implementation class by populating the *payment_gateway_configure* table. If your cartridge class name is *com.edocs.ps.MyCreditCardCartridge*, and you want to name it “customCCardCartridge”, use:

   insert into payment_gateway_configure(GATEWAY,PAYMENT_TYPE,CARTRIDGE_CLASS)values(‘customCCardCartridge’, ‘ccard’, ‘com.edocs.ps.MyCreditCardCartridge’);

8. When you go to payment settings of Command Center and configure a DNN for your credit card cartridge, the JSP page will read the list of available cartridges from this table and allow you to select one of them.

9. After you finish all the preceding steps, you should create a DDN, configure a cartridge for it and then make the payments from UI.

15 Miscellaneous Customization

Avoiding Paying a Bill More Than Once

By default, ePayment Manager allows a bill to be paid more than once. If you want to ensure that a bill can only be paid once, you need to add a unique key constraint on the *bill_id* field of the *check_payments* table. You can run *PAYMENT_HOME/db/set_unique_bill_id.sql* to set the unique constraint. Note: The *bill_id* in Payment is the same as the doc ID in eStatement Manager.

If a customer tries to pay a bill that has already been paid (either from the UI or by a previously scheduled recurring payment) after the unique key constraint has been added, the customer will receive an error message saying that the bill has been already paid. If the bill is paid from the UI and a recurring payment tries to pay it again, the payment will fail and an email notification message will be sent to the customer (if recurring payments are configured for that email notification).

Adding this constraint won't prevent a customer from making a payment using a bill ID. For example, a customer can still make a payment directly from the Make Check Payment link, which allows them make a payment without specifying a bill.

The unique key constraint only informs a customer that the bill has been paid when they try to pay a bill that has already been paid. If you want to provide additional features, such as disabling the payment button when the bill has already been paid, you must query the database to get that information. Be careful when adding extra functions, because performing additional database queries can affect ePayment Manager performance. Make sure the proper index has been created if you plan to create a new query.

Handling Multiple Payee ACH Accounts

By default, ePayment Manager only allows one payee (biller) ACH account per DDN, which is limited by Payment Settings. However, some billers may have multiple ACH accounts and their users will usually choose to pay to one of the ACH accounts when scheduling a payment. The way that the user chooses the ACH account to pay with can be based on some business rules added to the JSP. The rest of this section describes a solution to this problem.

The assumptions for this solution are:

- All ACH accounts are at the same bank, which means they have the same immediate origination and immediate destination but different company name and company Id.
- The business logic elements required to route the payment transaction to one ACH account versus another is available or can be made available in the Web application and in the execution context of a payment plugin.

We also assume there are *N* ACH accounts and there is one DDN for this biller. We call this DDN the "Real DDN". Here are the steps you need to go through:

1. Create a real DDN. You use this real DDN to configure Payment Settings for one of the ACH accounts.

2. Create virtual DDNs: Create *N* – 1 virtual DDNs, where each of their Payment Settings is configured to one of the *N* – 1 ACH accounts, respectively. Make sure the immediate origination and immediate destination are the same for all N DDNs but their company name and company ID are different.

Note: No indexer jobs run against these virtual DDNs; they are used solely for payment purposes.

3. Customize the UI: Your UI should employ some business logic to determine which DDN (effectively, ACH account) the payment transaction is to be entered against and set the payee ID of the payment to that DDN.

4. Run the pmtCheckSubmit Job: Configure a single pmtCheckSubmit job under the real DDN and configure it to pull payments from the all the *N* –1 virtual DDNs in addition to the real DDN. The payments from the same DDN will be under same batch.

5. Run the pmtCheckUpdate Job: pmtCheckUpdate processes the ACH return file. Since return files will include returns from all DDNs and the pmtCheckUpdate job can process these returns, we only need to create one pmtCheckUpdate job under the real DDN to process all the returned transactions (even though the returns may belong to other virtual DDNs).

6. Run the ePayment Manager pmtRecurringPayment Job: A single recurring payment job configured with the real DDN is required. A Recurring Payment plugin is required to execute the same logic as in scheduled payment; that is, apply the business rules to determine which DDN (effectively, ACH account) the recurring payment should be applied against. You should override the plug-ins `preSchedulePayment()` method for this purpose.

7. Change the pmtPaymentReminder Job setting: Six payment reminders, one per DDN, must be configured.

8. Run the pmtARIntegrator Job: The *AR_Query.xml* file is an XML definition of the database query that queries the ePayment Manager tables to build the default A/R file. The default query must be customized to include the virtual DDNs. Since the query is using the DDN reference numbers, you must pass that info into the query through one of these:

- Directly hard code the DDN references numbers in the query, though this is risky in the sense that if the DDN is re-created, your query will fail.
- Extend the `SampleARIntegrator` and overwrite the `getMap()` method and use *com.edocs.payment.util.DDNUtil* to find out the DDN reference number of a DDN, then set it as a "?" parameter used by the query. In this solution, the DDN names are hard coded but not the DDN reference numbers.
- Pass in the names of virtual DDNs as a flexible job configuration parameter from the job UI. The getMap() method can then parse the parameter to get the list of virtual DDNs. This method is recommended.

9. Add support for the ACH Prenote: If you are using ACH prenote, then you must create pmtSubmitEnroll, pmtConfirmEnroll and pmtNotifyEnroll jobs for each virtual DDN, which means you will get *N* prenote ACH files. pmtSubmitEnroll cannot aggregate prenotes from different DDNs into one.

Index