Payment Developer's Guide

V4.5
Document ID: EDXG-06-4.5-01
Data Published: 8.30.2004

edocs, Inc., One Apple Hill Dr., Natick, MA 01760

Printed in the USA.

Table of Contents

1 Preface 5

2 Introduction 9

Architecture of Payment 9

Major Payment Beans 11

3 Recurring Payments 15

Recurring Payment UI 17

Recurring payment – back end job: 23

Recurring Payment Scheduling 28

Recurring payment FAQ 31

4 Payment Plug-ins 35

Plug-In Customization 35

ACH Check Submit Plug-in 35

VeriSign Credit Card Payment Plug-in 37

Payment Reminder Plug-in 39

Recurring Payment Plug-in 40

5 Customizing Payment Template Files 43

Understanding the Payment Template Engine 43

Customizing Email Templates 45

Payment Reminder Template 46

Enrollment Notification Template 49

Recurring Payment Scheduled Notification Template 53

Payment Notification Template 55

Credit Card Expiration Notification Template 63

Customizing ACH Templates 64

Matching a Check in the ACH Return to the Database 67

6 Generating Accounts Receivables (A/R) Files 71

Customizing arQuery.xml 72

Query Case Study 73

Customizing arFlat_template.txt 75

Customizing arXML_template.xml 75

Customize arXML_template.xml and use XSLT to generate XML/flat AR file 76

Re-implement IARPaymentIntegrator 76

Select only check or credit card payments 77

Compiling and packaging a custom IARIntegrator 77

A/R Filenames 78

Single Payment Type 78

7 Packaging Payment Custom Code 79
8 Debugging Payment 81
Viewing WebLogic Logs 81
View logs from the Command Center: 81
Turning On the Payment Debug Flag 81
9 Plug-in Sample Code 83
AchCheckSubmitPlugIn.java 83
PaymentReminderPlugIn.java 85
RecurringPaymentPlugIn.java 86
VerisignCreditCardSubmitPlugIn.java 87
AddendaCheckSubmitPlugIn.java 89
SampleRecurringPlugIn.java 91
10 Auditing 95
Jobs that are audited 95
UI actions that are audited 96
Example UI Audit Flow 96
Query Files 101
Running Audit Queries 102
Query Setup 103
Windows Configuration 104
UNIX Configuration 104
Running the Queries in Windows 105
MSSQL 105
Oracle 106
DB2 107
Running the Queries in UNIX 108
Oracle 108
DB2 109
Audit Database 110
Modified Tables 110
New Tables 111
Audit Table Constants 112
Job Name Entries 112
11 Implementing A Custom Payment Cartridge 115
Demonstration Cartridge 115
Implementing Custom Credit Card Cartridge 115
12 Miscellaneous Customization 119
Avoiding paying a bill more than once 119
Handling multiple payee ACH accounts 119
13 Index 123

Preface

About Customer Self-Service and Payment™

edocs Payment™ is the electronic payment solution that decreases payment processing costs, accelerates receivables and improves operational efficiency. edocs Payment 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.

About This Guide

This guide describes the tasks required to develop an application to use Payment.

This guide is intended for the application developer and those involved in the process of designing a Payment application.

This guide assumes you have:

- Installed and configured Payment
- Know XML structure and syntax
- Undertand J2EE: JSP, HTML, Struts and Tiles

Related Documentation

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

Print Document	Description
Installation Guide	How to install Payment for your application and configure it in a distributed environment.
Payment Administration Guide	How to set up and run a live edocs application in a J2EE environment.
Payment Designer Guide	How to design your payment architecture.

Obtaining edocs Software and Documentation

You can download edocs software and documentation directly from Customer Central at https://support.edocs.com/. After you log in, click on the Downloads button on the left. When the next page appears, you will see a table displaying all of the available downloads. To search for specific items, select the Version and/or Category and click the Search Downloads button. If you download software, an email from edocs Technical Support will automatically be sent to you (the registered owner) with your license key information.

If you received an edocs product installation CD, load it on your system and navigate from its root directory to the folder where the software installer resides for your operating system. You can run the installer from that location, or you can copy it to your file system and run it from there. The product documentation included with your CD is in the Documentation folder located in the root directory. The license key information for the products on the CD is included with the package materials shipped with the CD.

If You Need Help

Technical Support is available to customers who have an active maintenance and support contract with edocs. Technical Support engineers can help you install, configure, and maintain your edocs application.

This guide contains general troubleshooting guidelines intended to empower you to resolve problems on your own. If you are still unable to identify and correct an issue, contact Technical Support for assistance.

Information to Provide

Before contacting edocs Technical Support, try resolving the problem yourself using the information provided in this guide. If you cannot resolve the issue on your own, be sure to gather the following information and have it handy when you contact technical support. This will enable your edocs support engineer to more quickly assess your problem and get you back up and running more quickly.

Please be prepared to provide Technical Support the following information:

Contact information:

- Your name and role in your organization.
- Your company's name
- Your phone number and best times to call you
- Your e-mail address

Product and platform:

- In which edocs product did the problem occur?
- What version of the product do you have?

- What is your operating system version? RDBMS? Other platform information?

Specific details about your problem:

- Did your system crash or hang?
- What system activity was taking place when the problem occurred?
- Did the system generate a screen error message? If so, please send us that message. (Type the error text or press the Print Screen button and paste the screen into your email.)
- Did the system write information to a log? If so, please send us that file.
- How did the system respond to the error?
- What steps have you taken to attempt to resolve the problem?
- What other information would we need to have (supporting data files, steps we'd need to take) to replicate the problem or error?

Problem severity:

- Clearly communicate the impact of the case (Severity I, II, III, IV) as well as the Priority (Urgent, High, Medium, Low, No Rush).
- Specify whether the problem occurred in a production or test environment.

Contacting edocs Technical Support

You can contact Technical Support online, by email, or by telephone.

edocs provides global Technical Support services from the following Support Centers:

US Support Center
Natick, MA
Mon-Fri 8:30am – 8:00pm US EST
Telephone: 508-652-8400

Europe Support Center
London, United Kingdom
Mon-Fri 9:00am – 5:00 GMT
Telephone: +44 20 8956 2673

Asia Pac Rim Support Center
Melbourne, Australia
Mon-Fri 9:00am – 5:00pm AU
Telephone: +61 3 9909 7301

Customer Central
https://support.edocs.com

Email Support
mailto:support@edocs.com

Escalation Process

edocs managerial escalation ensures that critical problems are properly managed through resolution including aligning proper resources and providing notification and frequent status reports to the client.

edocs escalation process has two tiers:

1. **Technical Escalation -** edocs technical escalation chain ensures access to the right technical resources to determine the best course of action.

2. **Managerial Escalation** - All severity 1 cases are immediately brought to the attention of the Technical Support Manager, who can align the necessary resources for resolution. Our escalation process ensures that critical problems are properly managed to resolution, and that clients as well as edocs executive management receive notification and frequent status reports.

By separating their tasks, the technical resources remain 100% focused on resolving the problem while the Support Manager handles communication and status.

To escalate your case, ask the Technical Support Engineer to:

1. Raise the severity level classification.

2. Put you in contact with the Technical Support Escalation Manager.

3. Request that the Director of Technical Support arrange a conference call with the Vice President of Services.

4. Contact VP of Services directly if you are still in need of more immediate assistance.

1

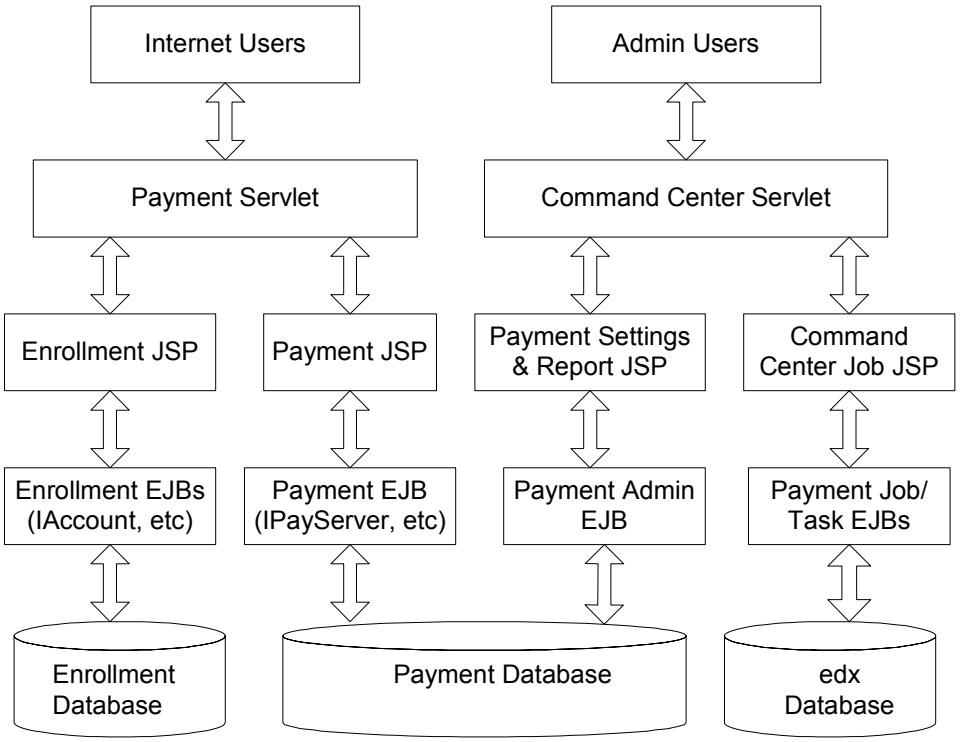
Introduction

Architecture of Payment

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

The following diagram shows an overview of the J2EE architecture of Payment:



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 Payment user JSP pages can be categorized into two groups:

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

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

There are also Payment batch jobs that run inside the command center. For a list and description of Payment jobs, refer to the *Administration Guide*.

Major Payment Beans

The following tables describe the major Payment 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 Payment 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 Payment 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 Payment 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-payment.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.

2

Recurring Payments

Payment's recurring payment feature is a very complicated process that involves a great deal of business logics. 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.

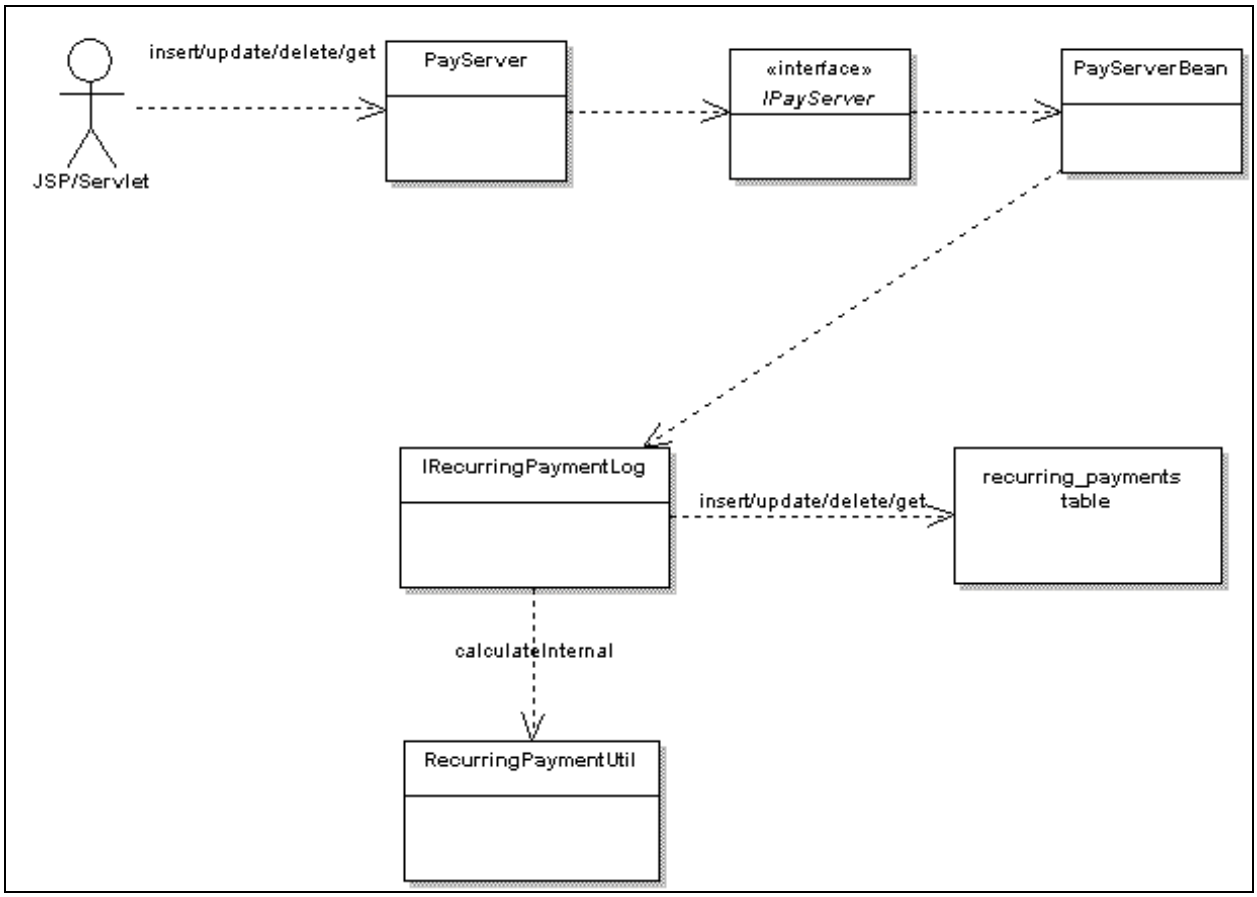
Column Name	Comment
LAST_PROCESS_TIME	Records the date of the last time the recurring payment job ran. During the recurring job synchronization process, Payment retrieves bills between LAST_PROCESS_TIME and current time to avoid retrieving duplicated bills and so, improve the performance. Note, previous versions of Payment 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 the command center.

Recurring Payment UI

This section discusses the actions of the recurring payment UI.

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

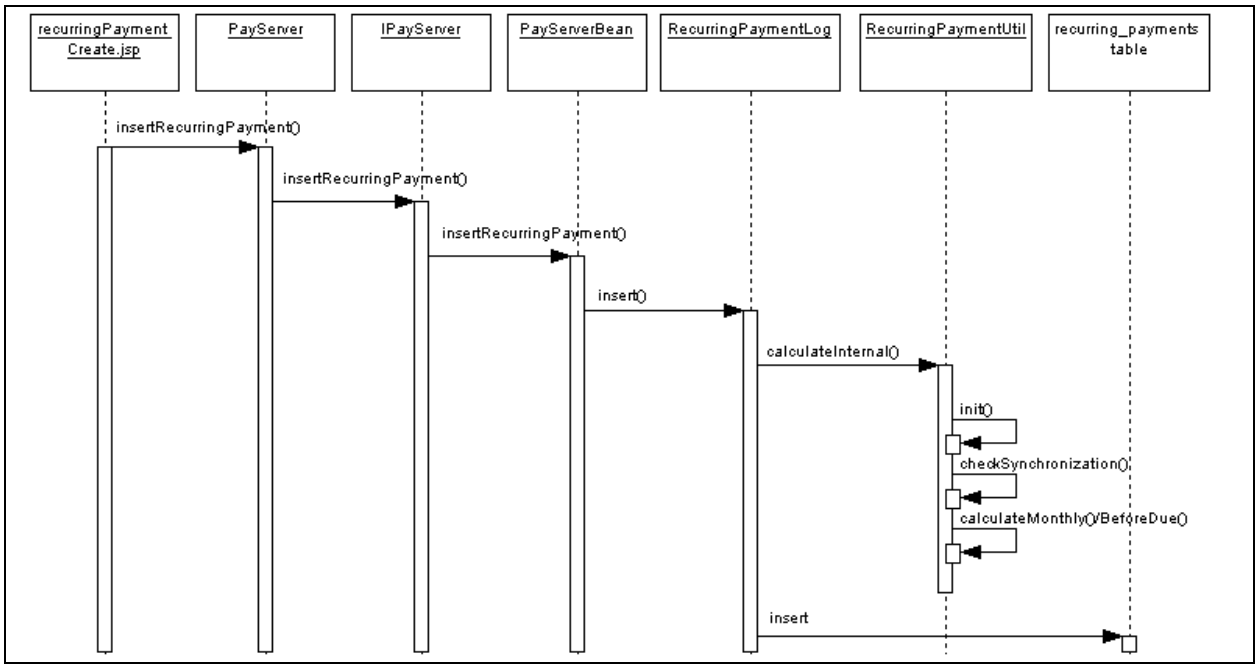
The following UML diagram illustrates the objects involved in the process.



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 from UI

The following sequence diagram demonstrates what happens when a recurring payment is inserted into database using the UI:



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: in Payment 4.2, the recurring payment UI has been enhanced to check 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:

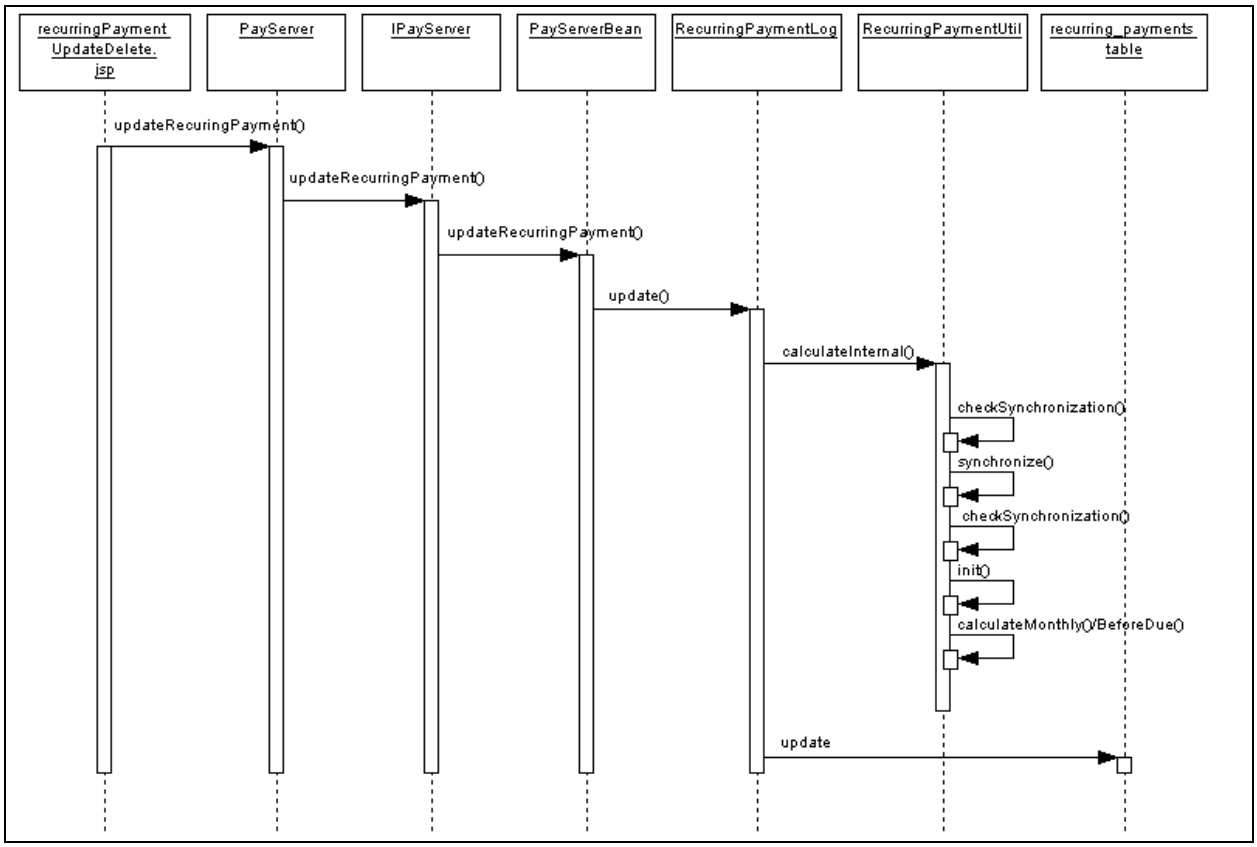
Day_of_pay_interval	Start_date	Next_pay_date
1	Sep 10	Oct 1
10	Sep 10	Sep 10
15	Sep 10	Oct 15
31	Sep 10	Sep 30

   - 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 from the UI

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.



The next section explains `RecurringPaymentUtil.calculateInternal()`. This method to 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.

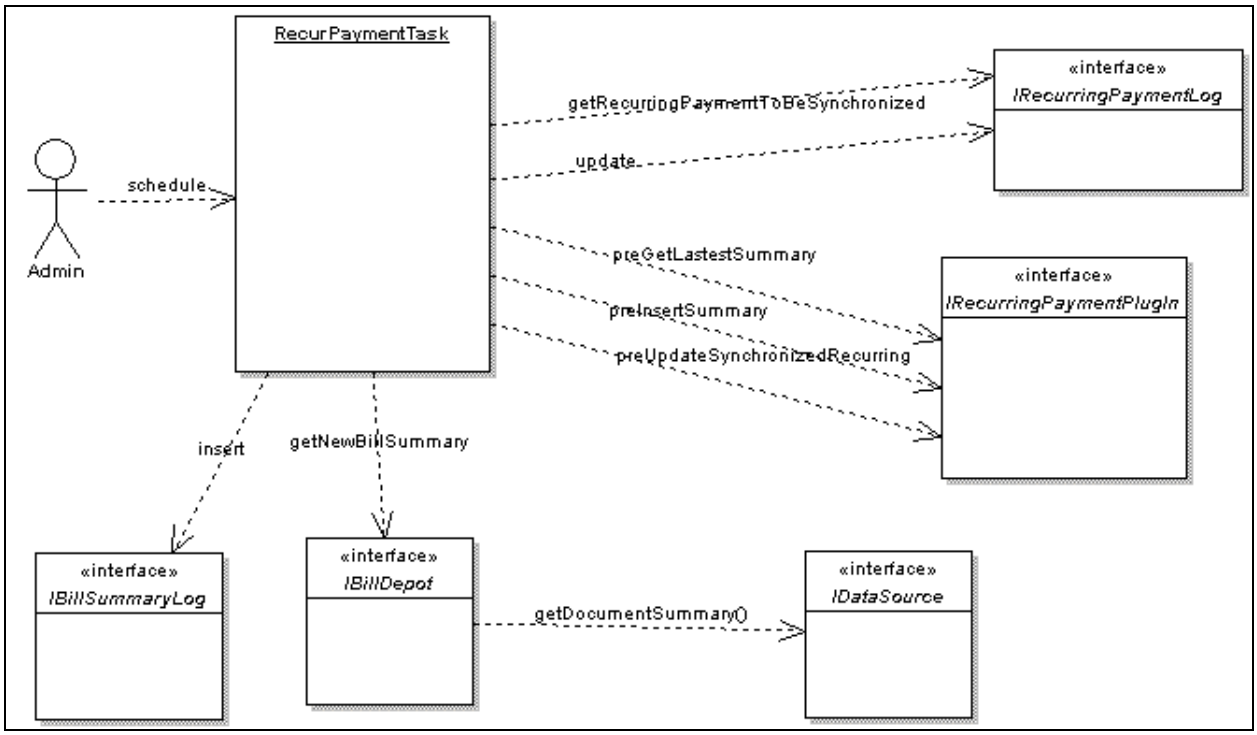
   Otherwise, set the status of the recurring payment to "active" and set its *next_pay_date* to the proposed next pay date.

Recurring payment – back end job:

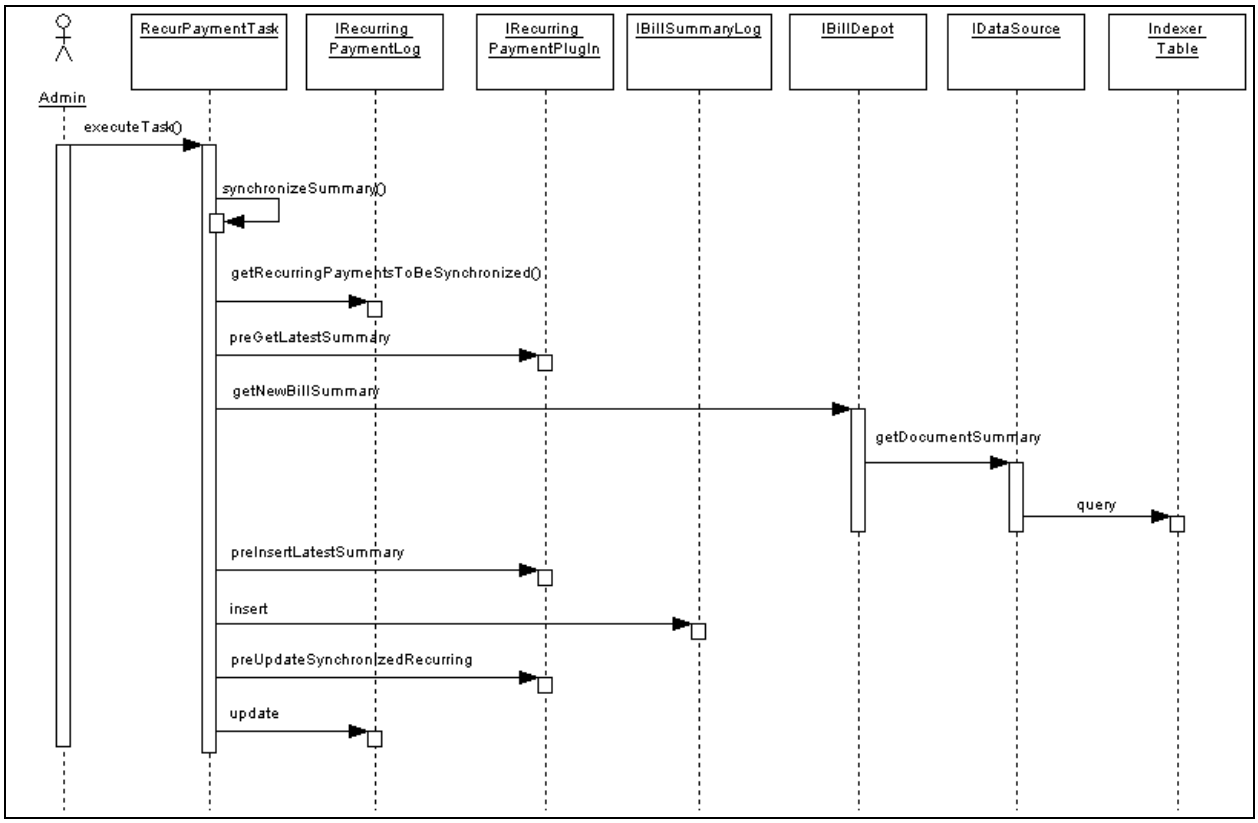
The pmtRecurringPayment job gets bills from the command center and then schedules payments. The first process is called “synchronization” and the second process is called “scheduling”. In Payment version 4.2, these two processes are split into two separate tasks.

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 the command center. The following UML diagram illustrates this process:



The following diagram shows the synchronization:



The following steps explain synchronization:

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

2. RecurringPaymentTask.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 eaDirect". 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 `RecurPaymentTask.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.

        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.
     c. 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.

        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.

Recurring Payment Scheduling

1. Call `RecurPaymentTask.isValidBillSummar()` 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 `RecurPaymentTask.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 `RecurPaymentTask.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.

3. Call `RecurPaymentTask.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 `RecurPaymenTask.updateRecurringPaymentOnly()` and return. See step 5 for details about what this method does.

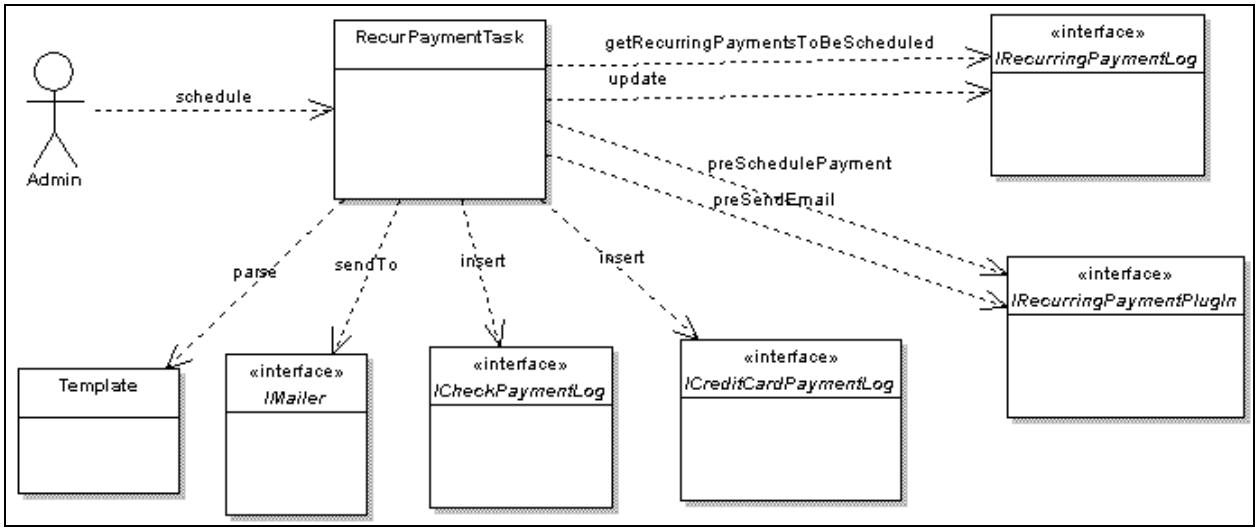
   - Call `IBillSummaryLog.insert()` to insert this new bill summary.

- If `IBillSummaryLog.insert()` throws DuplicateKeyException indicating that this bill is already in the database, so call `RecurPaymenTask.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:

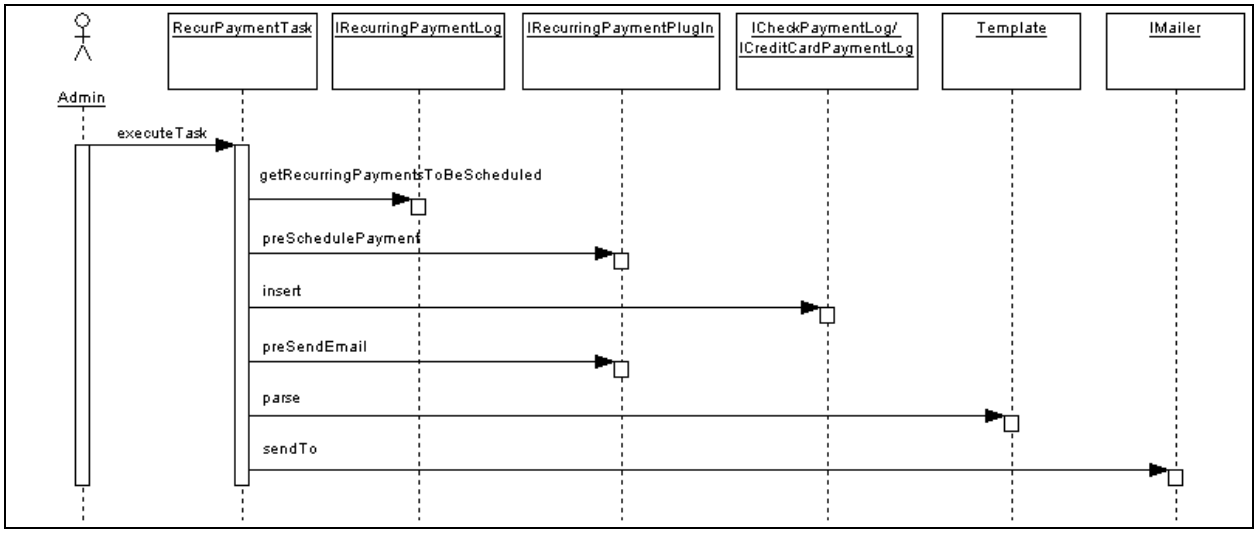
Column	Value
last_pay_date	In the case where the pay date is fixed, but amount is based on amount due, *last_pay_date* could be moved one *pay_interval* back if a scheduled payment is cancelled because a new bill arrives. Otherwise, *last_pay_date* will stay the same.
next_pay_date	*Next_pay_date* will 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 20 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. `RecurPaymentTask.execute()` is called when the job starts.

2. `RecurringPaymentTask.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 take 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. `RecurPaymentTask.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 gives PS a change 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 return to process next recurring payment; If not, the program will go 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

Column	Value
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.
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. `PaymentMailer.send()` is called to send emails. (`IMailer.sendTo()` was called for version 4.1.)

Recurring payment FAQ

This section answers a few common questions about recurring payment.

1. 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, Payment retrieves the latest bill for the account. Payment 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.

2. 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 billing system. For example, in the case of the command center, doc date can be used to indicate the chronological order of arriving bills. In the case of an external billing system, other dates such as statement date can be used for this purpose. When recurring payment synchronizes with the command center or other billing systems, it must 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.

3. Can recurring payment work with a billing system other than the command center?

Yes. Recurring payment assumes nothing specific to the command center 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.

4. 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 the command center, you must index the due date or some date equivalent to use as the due date.

5. 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 eaDirect?”. 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 rebill as a special case of rebill: rebill allows the same bill to appear more than once in a single billing period, but non rebill appears only once. The code and programming logic actually doesn’t distinguish between these two cases.

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

   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 the command center for each recurring payment. This can be especially true if you are talking with a billing system other than the command center 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.

7. 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 eaDirect” 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.

8. 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.

9. 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.

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

    No, you can only pay from one payment account for each recurring payment.

11. 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.

12. 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.

13. 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 with 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 or 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.

14. Will a single recurring payment failure fail the whole recurring payment job?

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

15. What is bill id?

    It's a unique id used to identify each bill. In the command center, it is the doc id.

16. 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 the command center. Payment 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 Payment also had time information as part of the last process time, but as of Payment 40, the last process time only contains date information (because the doc date only contains date information).

17. 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 eaDirect" 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 eaDirect" is set to "after current bill is scheduled", the second indexed bill will be ignored during next round of synchronization.

3

Payment Plug-ins

Plug-In Customization

The Payment plug-in is a callback, which allows you to add code to extend the functionality of Payment. There are four 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

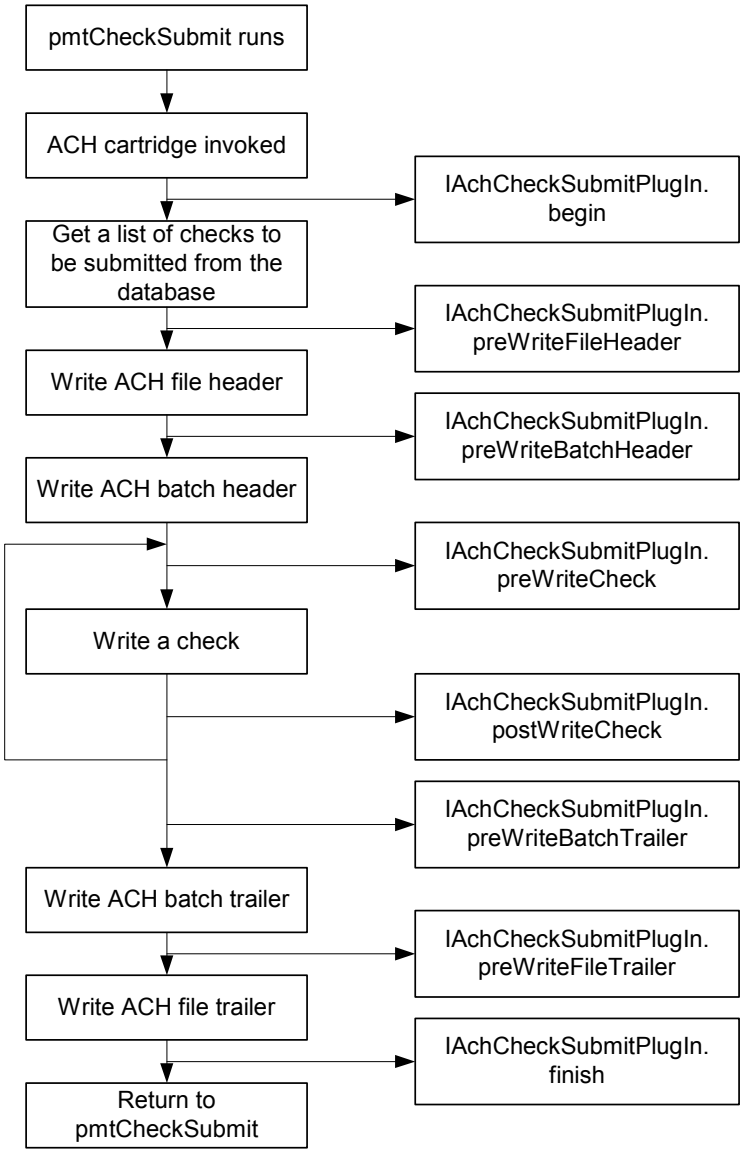
For each plug-in, Payment provides a default implementation. We recommend that you derive your plug-in from the default implementation to ensure that future updates to the plug-in will not break your code. The plug-ins and sample code are provided in *Sample Plugin Code* on page 83.

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:



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 Payment 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 79 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

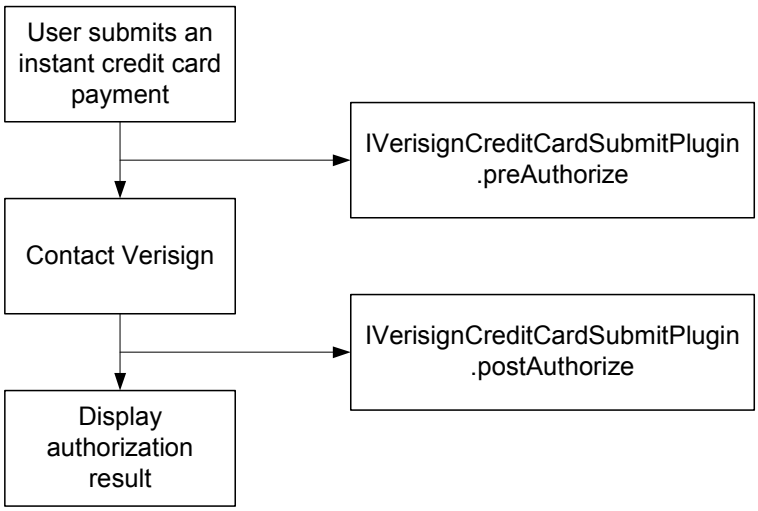
You can use the pmtCheckSubmit plug-in to write addenda records for ACH. The implementation called `AddendaCheckSubmitPlugIn` gets the invoice information of a payment and writes them out as addenda records. Check this class in the JavaDoc for its implementation details, and then follow the steps in *Writing a Plug-in* on page 36 to write your own implementation.

VeriSign Credit Card Payment Plug-in

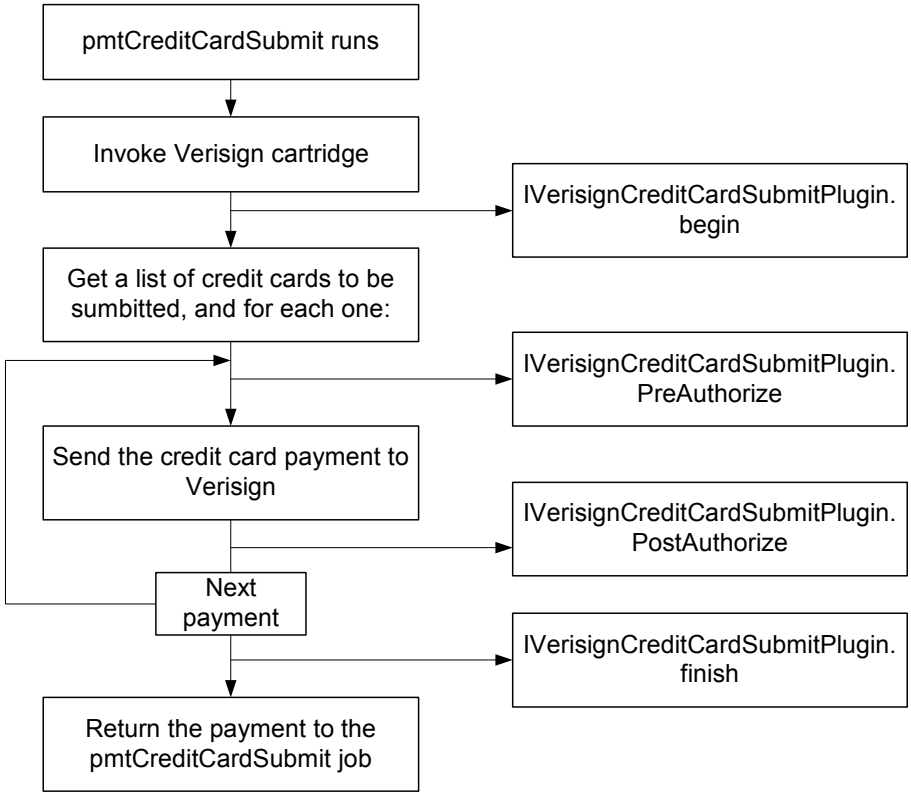
CreditCardSubmit Plug-in Overview

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:



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



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 *SDK: Customizing and Deploying Applications* document.
5. Change the Payment Settings of that DDN to use the new plug-in implementation.

Payment Reminder Plug-in

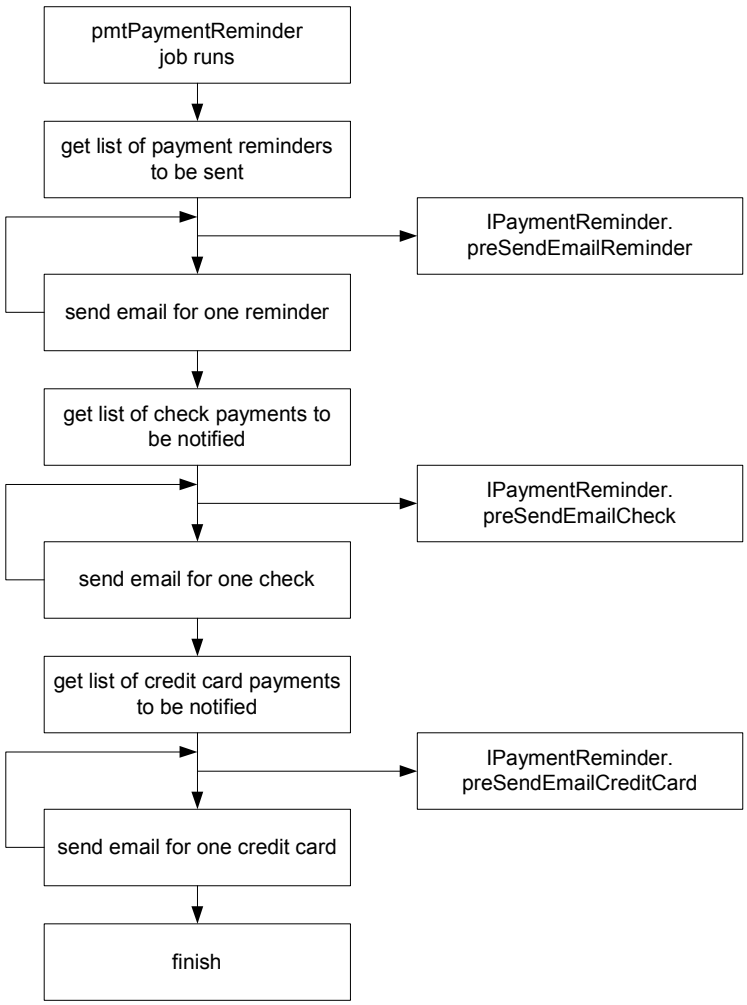
Payment Reminder Plug-in Overview

The payment 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:



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 the SDK: *Customizing and Deploying Applications* document.
5. Update the p*mtPaymentReminder* job configuration to use the new class.

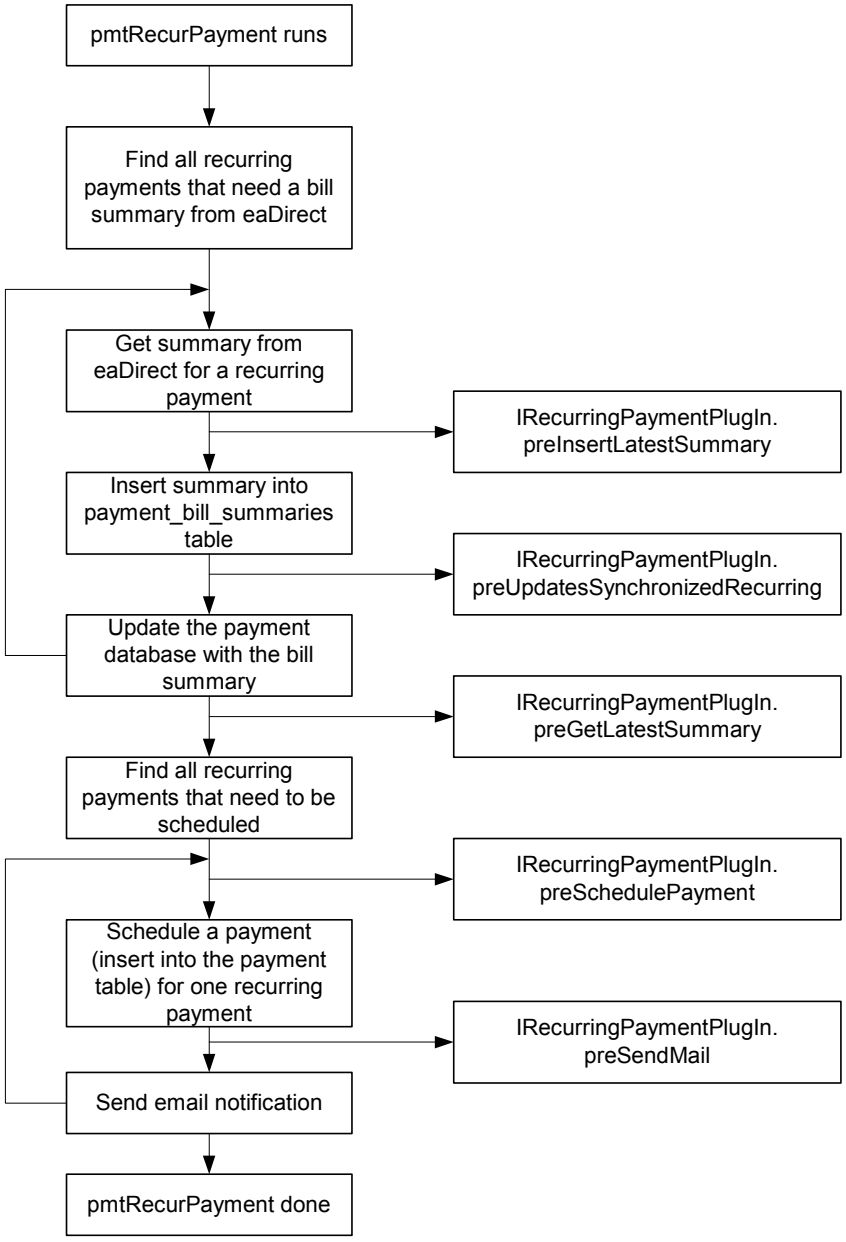
Recurring Payment Plug-in

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. See *Sample Recurring Plugin* on page 91 for more information.

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



Writing a Plug-in

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

1. Derive your implementation class from *RecurringPaymentPlugIn*.
2. Overwrite the method that you wish 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 *SDK: Customizing and Deploying Applications* document.

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.

4

Customizing Payment Template Files

Payment provides a template engine to generate Payment-wide text messages, such as emails, ACH files, and A/R files. This chapter describes how to use Payment templates to customize those text messages.

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 Payment 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 Payment hosted objects including their attributes and methods.

For more information about the Template class, see the Payment 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 Payment 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 payment 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 Payment 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 Email Templates

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

Separate email notification templates are used for:

Type of notification	Task that Specifies	Template File
Reminder to pay bills and the status of the checks	pmtPaymentReminder	*paymentReminder.txt*
Enrollment status	pmtNotifyEnroll	*motifyEnroll.txt*
Recurring payment was scheduled	pmtRecurPayment	*recurringNotify.txt*
Payment command center job status	All Payment jobs	*notifyPaymentTask.txt*
Credit card expiration	pmtCreditCardExpNotify	*CCExpNotify.txt*

For Unix, the default path to the email template files is *$PAYMENT_HOME/lib/payment_resources/*.
For Windows, it is:
%PAYMENT_HOME%\lib\payment_resources.

The email templates use a simple programming structure that works similar to JSP (but is **not** JSP). The 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 LOOP.

See the Payment JavaDoc for more information about the `Template` class.

Payment Reminder Template

Payment reminder messages are generated based on *PaymentReminder.txt*, which resides in $PAYMENT_HOME/*lib/payment_resources* (%PAYMENT_HOME%*lib\payment_resources* for Windows).

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

```
%<IF isRemind>%
Dear %reminder.getPayerId()%:
  This email is to remind you to pay your current
%reminder.getPayeeId()%'s
  bill. Please refer to this url to pay your bill:
  http://www.edocs.com.
  Thanks,
%</IF>%

%<IF isCheck>%
Dear user %check.getPayerId()%:

%<IF isPaid>%
%<IF! isAmtNegative>%
  Your check of $%check.getAmount()% has been paid on
%dateUtil.format("MMM dd yyyy", check.getPayDate())%.
%</IF!>%
%<IF isAmtNegative>%
  A credit of $%decimalUtil.absolute(check.getAmount())% has been
issued to your check account on %dateUtil.format("MMM dd yyyy",
check.getPayDate())%.
%</IF>%
%</IF>%

%<IF isReturned>%
%<IF! isAmtNegative>%
  Your check of $%check.getAmount()% has been returned. The error
message is:
%com.edocs.payment.cassette.ach.AchReturnCode.get(check.getTxnErr
Msg())%
%</IF!>%
%<IF isAmtNegative>%
  Your request to issue
$%decimalUtil.absolute(check.getAmount())% credit to your check
account has been rejected. The error message is:
%com.edocs.payment.cassette.ach.AchReturnCode.get(check.getTxnErr
Msg())%.
%</IF>%
%</IF>%
```

```
%<IF isFailed>%
  There is a problem to process your check. The error message is:
%check.getTxnErrMsg()%
%</IF>%

%<IF isCanceled>%
%<IF! isAmtNegative>%
  Your check of $%check.getAmount()% has been canceled by the
payment system because the check account is not valid. Please
check your enrollment information.
%</IF!>%
%<IF isAmtNegative>%
  Your request to issue
$%decimalUtil.absolute(check.getAmount())% credit to your check
account has been canceled by the payment system because the check
account is not valid. Please check your enrollment information.
%</IF>%
%</IF>%

%<IF isProcessed>%
%<IF! isAmtNegative>%
  Your check of $%check.getAmount()% has been sent to bank for
clearing.
%</IF!>%
%<IF isAmtNegative>%
  Your request to issue
$%decimalUtil.absolute(check.getAmount())% credit to your check
account has been sent to bank for clearing.
%</IF>%
%</IF>%

%</IF>%

%<IF isCCard>%
Dear user %creditcard.getPayerId()%:

%<IF isSettled>%
%<IF! isAmtNegative>%
  Your credit card payment of $%creditcard.getAmount()% has been
authorized successfully.
%</IF!>%
%<IF isAmtNegative>%
  Your request to reverse
$%decimalUtil.absolute(creditcard.getAmount())% to your credit
card has been authorized successfully.
%</IF>%
%</IF>%

%<IF isFailed>%
%<IF! isAmtNegative>%
  Your credit card payment of $%creditcard.getAmount()% failed
authorization.
  The error message is:  %creditcard.getTxnErrMsg()%
%</IF!>%
```

```
%<IF isAmtNegative>%
  Your request to reverse
$%decimalUtil.absolute(creditcard.getAmount())% to your credit
card failed authorization.
  The error message is:  %creditcard.getTxnErrMsg()%
%</IF>%
%</IF>%

%<IF isSystemFailure>%
%<IF! isAmtNegative>%
  Your credit card payment of $%creditcard.getAmount()% failed.
  The error message is:  %creditcard.getTxnErrMsg()%
%</IF!>%
%<IF isAmtNegative>%
  Your request to reverse
$%decimalUtil.absolute(creditcard.getAmount())% to your credit
card failed.
  The error message is:  %creditcard.getTxnErrMsg()%
%</IF>%
%</IF>%

%<IF isCanceled>%
%<IF! isAmtNegative>%
  Your credit card payment $%creditcard.getAmount()% has been
canceled by the payment system because the account is invalid.
Please check your enrollment information.
%</IF!>%
%<IF isAmtNegative>%
  Your request to reverse
$%decimalUtil.absolute(creditcard.getAmount())% to your credit
card has been canceled by the payment system because the account
is invalid. Please check your enrollment information.
%</IF>%
%</IF>%

%</IF>%
```

The following table describes the payment reminder template variables:

Variable	Type	Description
`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 (isFailedAuthorize).
`isPaid`	Boolean	True means the check has been paid or cleared.
`isProcessed`	Boolean	True means the check has been processed.

Variable	Type	Description
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.
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 *enrollNotify.txt*:

```
Dear %checkAccount.getUserId()%:

%<IF isACH>%
%<IF success>%
  Your payment account %checkAccount.getAccountNumber()% has
been succesfully activated.
%</IF>%
%<IF! success>%
  There has been a problem activating your payment account
%checkAccount.getAccountNumber()%.
  The return reason code is: %errCode%
%</IF!>%
%</IF>%

%<IF isNOC>%
%<IF isC01>%
%<IF isAutoUpdate>%
  Your Bank Account Number has been changed.
  New Bank Account Number is: %newPaymentAccount%
  Old Bank Account Number was: %oldPaymentAccount%
%</IF>%
%<IF! isAutoUpdate>%
  Your current Bank Account Number is out of date.
  New Bank Account Number is: %newPaymentAccount%
  Current Bank Account Number is: %oldPaymentAccount%
  Please login to change your profile.
%</IF!>%
%</IF>%
```

```
%<IF isC02>%
%<IF isAutoUpdate>%
  Your Bank Routing Number has been changed.
  New Bank Routing Number is: %newRouting%
  Old Bank Routing Number was: %oldRouting%
%</IF>%
%<IF! isAutoUpdate>%
  Your current Bank Routing Number is out of date.
  New Bank Routing Number is: %newRouting%
  Current Bank Routing Number is: %oldRouting%
  Please login to change your profile.
%</IF!>%
%</IF>%
%<IF isC03>%
%<IF isAutoUpdate>%
  Your Bank Account Information has been changed.
  New Bank Account Number is: %newPaymentAccount%
  Old Bank Account Number was: %oldPaymentAccount%
  New Bank Routing Number is: %newRouting%
  Old Bank Routing Number was: %oldRouting%
%</IF>%
%<IF! isAutoUpdate>%
  Your current Bank Account Information is out of date.
  New Bank Account Number is: %newPaymentAccount%
  Current Bank Account Number was: %oldPaymentAccount%
  New Bank Routing Number is: %newRouting%
  Current Bank Routing Number is: %oldRouting%
  Please login to change your profile.
%</IF!>%
%</IF>%
%<IF isC05>%
%<IF isAutoUpdate>%
  Your Bank Account Information has been changed.
  Your new Bank Type is %newPaymentType%
  Your old Bank Type was %oldPaymentType%
%</IF>%
%<IF! isAutoUpdate>%
  Your current Bank Account Type is out of date.
  Your new Bank Type is %newPaymentType%
  Your current Bank Type is %oldPaymentType%
  Please login to change your profile.
%</IF!>%
%</IF>%
%<IF isC06>%
%<IF isAutoUpdate>%
  Your Bank Account Information has been changed.
  New Bank Account Number is: %newPaymentAccount%
  Old Bank Account Number was: %oldPaymentAccount%
  Your new Bank Type is %newPaymentType%
  Your old Bank Type was %oldPaymentType%
%</IF>%
%<IF! isAutoUpdate>%
  Your current Bank Account Information is out of date.
  New Bank Account Number is: %newPaymentAccount%
  current Bank Account Number is: %oldPaymentAccount%
  New Bank Type is %newPaymentType%
  Current Bank Type is %oldPaymentType%
```

```
  Please login to change your profile.
%</IF!>%
%</IF>%
%<IF isC07>%
%<IF isAutoUpdate>%
  Your Bank Account Information has been changed.
  New Bank Account Number is: %newPaymentAccount%
  Old Bank Account Number was: %oldPaymentAccount%
  New Bank Routing Number is: %newRouting%
  Old Bank Routing Number was: %oldRouting%
  Your new Bank Type is %newPaymentType%
  Your old Bank Type was %oldPaymentType%
%</IF>%
%<IF! isAutoUpdate>%
  Your current Bank Account Information is out of date.
  New Bank Account Number is: %newPaymentAccount%
  Current Bank Account Number is: %oldPaymentAccount%
  New Bank Routing Number is: %newRouting%
  Current Bank Routing Number is: %oldRouting% <>
  New Bank Type is %newPaymentType%
  Current Bank Type is %oldPaymentType%
  Please login to change your profile.
%</IF!>%
%</IF>%
%</IF>%

%<IF isCDP>%
%<IF success>%
  Your payment account %checkAccount.getAccountNumber()% has
been succesfully activated.
%</IF>%
%<IF! success>%
      There has been a problem activating your payment account
%checkAccount.getAccountNumber()%. Please contact your
customer service representative for further assistance.
%</IF!>%
%</IF>%
```

This template is used for both ACH and Checkfree CDP. The text between `%<IF isACH>%` and the corresponding `%</IF>%` is for ACH. The text between `%<IF isCDP>%` and the corresponding `%</IF>%` 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. Payment 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 Scheduled Notification Template

When recurring payment schedules a payment, email notification messages are generated from the template file *recurringNotify.txt*:

```
Dear %recurringPayment.getPayerId()%,
%<IF isPaymentScheduled>
  %<IF isCheck>%

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

The check amount is $%payment.getAmount()%. The pay date is
%dateUtil.format("MM/dd/yyyy", payment.getPayDate())%.

  %</IF>%

  %<IF isCCard>%

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

The check amount is $%payment.getAmount()%. The pay date is
%dateUtil.format("MM/dd/yyyy", payment.getPayDate())%.

  %</IF>%^M

        You can update or cancel this transaction following
this URL http://www.edocs.com.

%</IF>%

%<IF isPaymentNotScheduled>%

Dear %recurringPayment.getPayerId()%,

        This email is to inform you that your recurring
payment scheduled on %dateUtil.format("MM/dd/yyyy",
recurringPayment.getNextPayDate())% is not made as requested.
Please contact your biller for detail.

%</IF>%

%<IF isLessPayment>%

Dear %recurringPayment.getPayerId()%,

        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.

%</IF>%

%<IF isAlreadyPaid>%

Dear %recurringPayment.getPayerId()%,

        The bill, due on %dateUtil.format("MM/dd/yyyy",
recurringPayment.getBillDueDate())%, is not paid

by recurring payment because it is already been paid.

%</IF>%

%<IF isLastRecurringPayment>%
```

```
        This is the last payment from the recurring payment.
%</IF>%
%<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.
        Please contact your biller for further details.
%</IF>%
        Number of payments made until now is
%recurringPayment.getCurrNumPayment()%.
```

The recurring notification template variables are:

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.
`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.

Variable Name	Type	Description
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 *notifyPaymentTask.txt*:

```
%<IF isOK>%
  %taskName% was done without error at
%dateUtil.format("MM/dd/yyyy HH:mm:ss", currentTime)%.
%</IF>%
%<IF! isOK>%
  %taskName% was done with an error at
%dateUtil.format("MM/dd/yyyy HH:mm:ss", currentTime)%. The
error message is : %taskException.getMessage()%.
%</IF!>%

%<IF skipSynchronization>%

  As Skip SynchronizationTask setting is set to YES, The
Synchronization Task was skipped
%</IF>%

%<IF recurringPmtSyncTask>%
 %<IF isDone>%
   Job Name : %jobName%
   Total Number of RecurringPayments to be synchronized :
%syncCount%
   Total Number of RecurringPayments that are synchronized
Successfully : %syncSuccessCount%
   Total Number of Recurring Payments that failed to
synchronize : %syncFailureCount%.
 %</IF>%
%<IF! isDone>%
   Please look at the audit tables for detail.

   Job Name : %jobName%
   Total Number of RecurringPayments to be synchronized :
%syncCount%
   Total Number of RecurringPayments that are synchronized
Successfully : %syncSuccessCount%
   Total Number of Recurring Payments that failed to
synchronize : %syncFailureCount%.

 %</IF!>%
%</IF>%
```

```
%<IF recurringPmtSchedulerTask>%
 %<IF isDone>%
   Job Name : %jobName%
   Total Number of RecurringPayments to be scheduled :
%scheduleCount%
   Total Number of RecurringPayments that are scheduled
Successfully : %scheduleSuccessCount%
   Total Number of Recurring Payments that failed to be
scheduled : %scheduleFailureCount%.
     %<IF isDecryptFailed>%

   Total Number of Recurring Payments cancelled due to
decryption failure : %CancelCount%

     %</IF>%
 %</IF>%
%<IF! isDone>%
   Please look at the audit tables for detail.

   Job Name : %jobName%
   Total Number of RecurringPayments to be scheduled  :
%scheduleCount%
   Total Number of RecurringPayments that are scheduled
Successfully : %scheduleSuccessCount%
   Total Number of Recurring Payments that failed to be
scheduled  : %scheduleFailureCount%.
     %<IF isDecryptFailed>%

   Total Number of Recurring Payments cancelled due to
decryption failure : %CancelCount%

     %</IF>%

 %</IF!>%
%</IF>%

%<IF paymentReminderTask>%
 %<IF isDone>%
   Job Name : %jobName%
   Total Number of Good Check Payment notifications :
%goodCheckPaymentsCount%
   Total Number of Check Payment notifications failed due to
decryption failure : %badCheckPaymentsCount%

   Total Number of Good CreditCard Payment notifications :
%goodCCPaymentsCount%
   Total Number of CreditCard Payment notifications failed due
to decryption failure : %badCCPaymentsCount%
 %</IF>%
%</IF>%

%<IF CreditCardExpNotifyTask>%
 %<IF isDone>%
   Job Name : %jobName%

   Total Number of CreditCard expiration notifications to be
processed : %ccexpNotifyCount%
```

```
   Total Number of CreditCard expiration notifications that
are processed Successfully : %ccexpNotifySuccessCount%
   Total Number of CreditCard expiration notifications that
are failed : %ccexpNotifyFailureCount%

   Total Number of Good CreditCard notifications :
%goodCCAccountCount%
   Total Number of Bad CreditCard notifications  :
%badCCAccountCount%
 %</IF>%
%</IF>%

%<IF CheckSubmitTask>%
%<IF isDone>%
   Job Name : %jobName%
   %<IF isHoliday>%
         This job was not run since today
(%dateUtil.format("MM/dd/yyyy", todayDate)%) is a holiday.
   %</IF>%
   %<IF isDecryptFailed>%
         While running the job, there were account decryption
failures.
   %</IF>%
 %</IF>%
%</IF>%

%<IF SubmitEnrollTask>%
%<IF isDone>%
   Job Name : %jobName%
   %<IF isHoliday>%
         This job was not run since today
(%dateUtil.format("MM/dd/yyyy", todayDate)%) is a holiday.
   %</IF>%

   %<IF isDecryptFailed>%
         While running the job, there were account decryption
failures.
   %</IF>%
 %</IF>%
%</IF>%

%<IF CreditCardSubmitTask>%
%<IF isDone>%
   Job Name : %jobName%
   %<IF isDecryptFailed>%
         While running the job, there were account decryption
failures.
   %</IF>%
 %</IF>%
%</IF>%
```

pmtCreditCardExpNotify Variables

The payment notification template variables related to pmtCreditCardExpNotifiy are:

Variable	Value type	Description
CreditCardExpNotifyTask	String	Identifies the credit card expiration notification task.
isDone	Boolean (true or false)	Identifies the job had done.
jobName	String	Identifies the job name.
ccexpNotifyCount	int	Total number of notifications to be made.
ccexpNotifySuccessCount	int	Successful number of accounts.
ccexpNotifyFailureCount	int	Failed number of accounts.
goodCCAccountCount	int	Number of good credit card accounts (due to decryption).
badCCAccountCount	int	Number of bad credit card accounts (due to decryption).

Example:

```
%<IF recurringPmtSyncTask>%
%<IF skipSynchronization>%

  As Skip SynchronizationTask setting is set to YES, The
Synchronization Task was skipped
%</IF>%
 %<IF isDone>%
   Job Name : %jobName%
   Total Number of RecurringPayments to be synchronized :
%syncCount%
   Total Number of RecurringPayments that are synchronized
Successfully : %syncSuccessCount%
   Total Number of Recurring Payments that failed to
synchronize : %syncFailureCount%.
 %</IF>%
%<IF! isDone>%
   Please look at the audit tables for detail.

   Job Name : %jobName%
   Total Number of RecurringPayments to be synchronized :
%syncCount%
   Total Number of RecurringPayments that are synchronized
Successfully : %syncSuccessCount%
   Total Number of Recurring Payments that failed to
synchronize : %syncFailureCount%.

 %</IF!>%
%</IF>%
```

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.
syncSuccessCount	int	Successful number of synchronized accounts.
syncFailureCount	int	Number of failed of synchronized accounts.

Example:

```
%<IF recurringPmtSyncTask>%
%<IF skipSynchronization>%
  As Skip SynchronizationTask setting is set to YES, The
Synchronization Task was skipped
%</IF>%
 %<IF isDone>%
   Job Name : %jobName%
   Total Number of RecurringPayments to be synchronized :
%syncCount%
   Total Number of RecurringPayments that are synchronized
Successfully : %syncSuccessCount%
   Total Number of Recurring Payments that failed to
synchronize : %syncFailureCount%.
 %</IF>%
%<IF! isDone>%
   Please look at the audit tables for detail.
   Job Name : %jobName%
   Total Number of RecurringPayments to be synchronized :
%syncCount%
   Total Number of RecurringPayments that are synchronized
Successfully : %syncSuccessCount%
   Total Number of Recurring Payments that failed to
synchronize : %syncFailureCount%.
 %</IF!>%
%</IF>%
```

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.

Recurring Scheduler Variable	Type	Description
jobName	String	To identify the job name.
scheduleCount	Int	Total number of accounts to be scheduled
scheduleSuccessCount	Int	Successful number of scheduled accounts
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 recurringPmtSyncTask>%
%<IF skipSynchronization>%
  As Skip SynchronizationTask setting is set to YES, The
Synchronization Task was skipped
%</IF>%
 %<IF isDone>%
   Job Name : %jobName%
   Total Number of RecurringPayments to be synchronized :
%syncCount%
   Total Number of RecurringPayments that are synchronized
Successfully : %syncSuccessCount%
   Total Number of Recurring Payments that failed to
synchronize : %syncFailureCount%.
 %</IF>%
%<IF! isDone>%
   Please look at the audit tables for detail.
   Job Name : %jobName%
   Total Number of RecurringPayments to be synchronized :
%syncCount%
   Total Number of RecurringPayments that are synchronized
Successfully : %syncSuccessCount%
   Total Number of Recurring Payments that failed to
synchronize : %syncFailureCount%.
 %</IF!>%
%</IF>%
```

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.

Reminder Variable	Type	Description
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>%
   Job Name : %jobName%
   Total Number of Good Check Payment notifications :
%goodCheckPaymentsCount%
   Total Number of Check Payment notifications failed due to
decryption failure : %badCheckPaymentsCount%

   Total Number of Good CreditCard Payment notifications :
%goodCCPaymentsCount%
   Total Number of CreditCard Payment notifications failed due
to decryption failure : %badCCPaymentsCount%
 %</IF>%
%</IF>%
```

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>%
 Job Name : %jobName%

 Total Number of CreditCard expiration notifications to be processed : %ccexpNotifyCount%
 Total Number of CreditCard expiration notifications that are processed Successfully : %ccexpNotifySuccessCount%
 Total Number of CreditCard expiration notifications that are failed : %ccexpNotifyFailureCount%

 Total Number of Good CreditCard notifications : %goodCCAccountCount%
 Total Number of Bad CreditCard notifications : %badCCAccountCount%
 %</IF>%
%</IF>%

pmtCheckSubmit Variables

Check Submit Variable	Type	Description
CheckSubmitTask	Boolean value (true or false)	Identifies the check submit 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.
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>%
   Job Name : %jobName%
   %<IF isHoliday>%
         This job was not run since today
(%dateUtil.format("MM/dd/yyyy", todayDate)%) is a holiday.
   %</IF>%
   %<IF isDecryptFailed >%
         While running the job, there were account decryption
failures.
   %</IF>%
 %</IF>%
%</IF>%
```

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.

Example:

```
%<IF SubmitEnrollTask>%
%<IF isDone>%
   Job Name : %jobName%
   %<IF isHoliday>%
         This job was not run since today
(%dateUtil.format("MM/dd/yyyy", todayDate)%) is a holiday.
   %</IF>%

   %<IF isDecryptFailed>%
         While running the job, there were account decryption
failures.
   %</IF>%
 %</IF>%
%</IF>%
```

Credit Card Expiration Notification Template

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

```
Dear %account.getUserId()%:

  This email is to remind you that your credit card which has
the account number %account.getShorttenedAccNumber()%,

  %<IF! accExpired>%

   is about to expire in %dateUtil.format("MMM yyyy",
account.getExpireDate ())%.

  %</IF!>%

  %<IF accExpired>%

  has expired in %dateUtil.format("MMM yyyy",
account.getExpireDate())%.

  %</IF>%


Please login to the Payment system and update the credit card
information.
```

```
Thanks
```

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

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 Payment.
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.

Global Variable Name	Type	Description
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.

Variable Name	Type	Description
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.

Payment 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 Payment 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 Payment. 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, Payment 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>
```

   and rename “target1” to “target” to enable using trace number as gateway payment ID.

5

Generating Accounts Receivables (A/R) Files

It is often necessary to synchronize the Payment system with a biller's A/R system. Payment usually needs to periodically send A/R files to a biller's A/R system, which includes the payments being made through Payment. The format of the file varies among billers. To support this function, Payment 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 Payment 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 *flexible_field_3* 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.txt*, 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 *flexible_field_3* 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 edocs XMLQuery technology. For details about this definition, see the *SDK: Content Access* document that is part of the command center SDK.

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 Payment 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 flexible_field_3 = '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 Payment payment tables. They both use *flexible_field_3* 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 *flexible_field_3* as an A/R flag, you can create an index for it to increase performance. Payment provides a script just for that purpose in *EDCSpay/db/create_ar_index.sql*. This script is not run when the Payment 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 '?'s 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 flexible_field_3 = '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`.

The `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>
```

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 Payment 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 Payment 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 Payment:

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 Payment 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 `flexible_field_3` 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.

See "How to compile/package Payment custom code".

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.

6

Packaging Payment Custom Code

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 Payment EAR files will access this JAR, and find the custom code. The Payment EAR files are merged into the command center EAR file as part of installation, so your custom code will also be seen by the command center.

To make this JAR file accessible by all the Payment 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 *SDK: Customizing and Deploying Applications* document that comes with the Command Center SDK. 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 Payment EAR files with the new *Payment_custom.jar*, using jar command.
4. Deploy the new Payment 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*.

7

Debugging Payment

First, follow the installation steps carefully to set up Payment. 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 the Command Center:

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

Turning On the Payment Debug Flag

If you have problems with executing payment 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 …
```

8

Plug-in Sample Code

This chapter lists the sample code for the job plug-ins, for:

Job	Plug-in Code
pmtPaymentReminder	*PaymentReminderPlugIn.java* on page 85
pmtCreditCardSubmit	*VerisignCreditCardSubmitPlugIn.java* on page 87
pmtCheckSubmit	*AchCheckSubmitPlugIn.java* on page 83 *AddendaCheckSubmitPlugIn.java* on page 89 shows an example implementation.
pmtRecurringPayment	*RecurringPaymentPlugIn.javaon* page 86 *SampleRecurringPlugIn.java* on page 91 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;
  }
}
```

9

Auditing

Payment 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 that are 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 that are audited

Lists successful and unsuccessful payments along with a reason code.

The UI actions that trigger an audit entry are listed below"

- 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

Example 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.

Column	Value	Description
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, and then 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	Tthis 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.
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).

Column	Value	Description
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.

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.

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.
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 supprt queries of the audit tables.

Windows SQL2000

getAuditDataByPid.bat
getAuditDataByPaymentId.bat
getAuditDataByAccount.bat

ORACLE

Windows

getAuditDataByAccount.bat
getAuditDataByAccount.sql
getAuditDataByPaymentId.bat
getAuditDataByPaymentId.sql
getAuditDataByPid.bat
getAuditDataByPid.sql
set_audit_isql_options.bat

Unix

getAuditInfoByAccount.sh
getAuditInfoByAccount.sql
getAuditInfoByPaymentId.sh
getAuditInfoByPaymentId.sql
getAuditInfoByPid.sh
getAuditInfoByPid.sql

DB2

Windows

getAuditDataByAccount.bat
getAuditDataByPaymentId.bat
getAuditDataByPid.bat
set_audit_isql_options.bat

Unix

getAuditDataByAccount.sh
getAuditDataByPaymentId.sh
getAuditDataByPid.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 Command Center/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.

Email Type	Description
4	Check status notification email.
5	Credit card status notification email.
6	Recurring payment cancelled email.
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

11

Implementing A Custom Payment 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 Payment JSP pages get the credit card information from the user and pass the information to the cartridge. After the card is processed, Payment updates Payment 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 Payment 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 Payment API to update this credit card back to Payment database (note, this is different from Instant payments, because Payment does not update the database).

6. Package your custom cartridge.

   If you are using Payment2.2 with WebSphere, you should package it into *Payment_client.jar* which is in the *lib* dir of each Payment EAR.

   If you are using Payment3.0 with WebLogic, you should package it into *Payment_custom.jar* which is in the *lib* directory of each Payment EAR.

7. Pre-populate Payment database.

   Tell Payment 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.

12 Miscellaneous Customization

Avoiding paying a bill more than once

By default, Payment 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 the command center.

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 to 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 Payment performance. Make sure the proper index has been created if you plan to create a new query.

Handling multiple payee ACH accounts

By default, Payment 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 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, there will be 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 Payment 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-in's `preSchedulePayment()` method for this purpose.

7. Change the Payment 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 Payment payment 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

A

ACH
  addenda records, 37
  customizing, 64
  individual ID, 44, 64
  individual name, 64
  plugin, 35
  return files, 67
  templates, 64
addenda records, 37

E

enrollment
  email template, 49

H

Help
  technical support, 6

P

Payment ear
  beans, 11
payments reminders
  template, 46
plugin
  creating for credit cards, 38
  for recurring payments, 40
  overview for credit cards, 37
  overview for reminders, 39
plug-in
  creating for ACH, 36
  overview for ACH, 35
pmtCheckSubmit
  bean, 12
  job email template, 62
  plugin, 35
pmtCheckUpdate
  bean, 13
pmtConfirmEnroll
  bean, 13
pmtCreditCardExpNotifiy
  user email template, 64
pmtCreditCardExpNotify
  job email template, 58, 61
pmtCreditCardSubmit
  bean, 12
pmtNotifyEnroll
  bean, 13
pmtPaymentReminder
  bean, 14
  job email template, 60
pmtRecurPayment
  bean, 14
  email template, 53
pmtRecurringPayment
  job email template, 59
  jobemail template, 59
pmtSubmitEnroll
  bean, 14
  job email template, 63

R

recurring payments
  email template, 53, 55
  plugin, 40
reminders
  plugin, 39