IBM Payment Manager Cassette Development Workshop

Payment Manager Introduction

Jim Reach
e-commerce
ISV Development

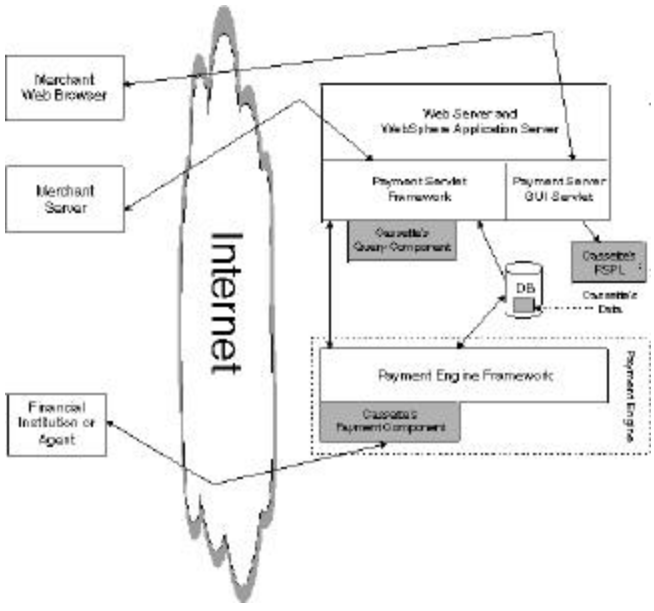
The Payment Manager...

- Manages the life cycle of electronic payments through cash register-like functionality
- Provides single programming and user interface for different payment instruments
- Is driven by and cooperates with business applications like
  - Catalog Systems
  - Fulfillment Systems
  - Accounting Systems.
- Is extendable to new payment technologies through new casettes with minimal change to application systems
- Is composed of several Java servlets and Java applications

The Platforms

- Windows/NT V4
- Solaris Version 2.6
- AIX Version 4.2.1, 4.3.2, or 4.3.3
- AS/400
- OS/390 coming soon (only with WebSphere Commerce Suite)

Architecture



Main Components

- Payment Servlet provides HTTP-based API and performs API-based queries
- Payment Engine maintains configuration and performs financial transactions
- User Interface Servlet provides HTML-based UI exclusively through the Payment Manager API
- A web server (IBM HTTP Server included)
- IBM WebSphere Application Server (included)
- A database (IBM DB2 Universal Database included)
- Cassettes
  - IBM SET Cassette
  - IBM CyberCash Cassette
  - Soon we will be adding yours.

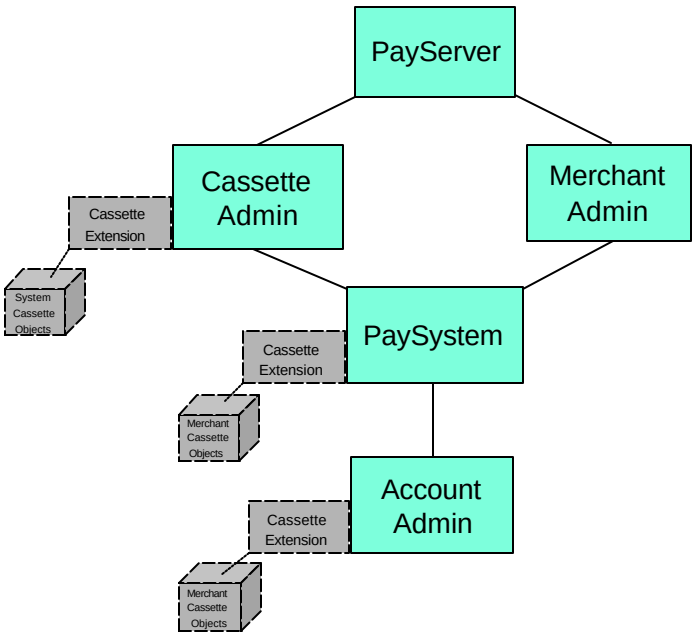
Overview of Framework

- Data Model
- Application Programming Interface
- Database Access
- Memory management
- Synchronization
- Parameter Validation and Conversion
- Thread Management
- Access control
- Scheduling background and timed tasks
- Protocol message management
- Event notification
- Exception Management
- Tracing Internal Events
- Logging Errors

The Data Model

- Administration objects
  - PayServer
  - CassetteAdmin
  - MerchantAdmin
  - PaySystem
  - AccountAdmin
  - Cassette-specific objects
    - Cassette scope
    - Merchant scope
- Financial objects
  - Order
  - Payment
  - Credit
  - Batch

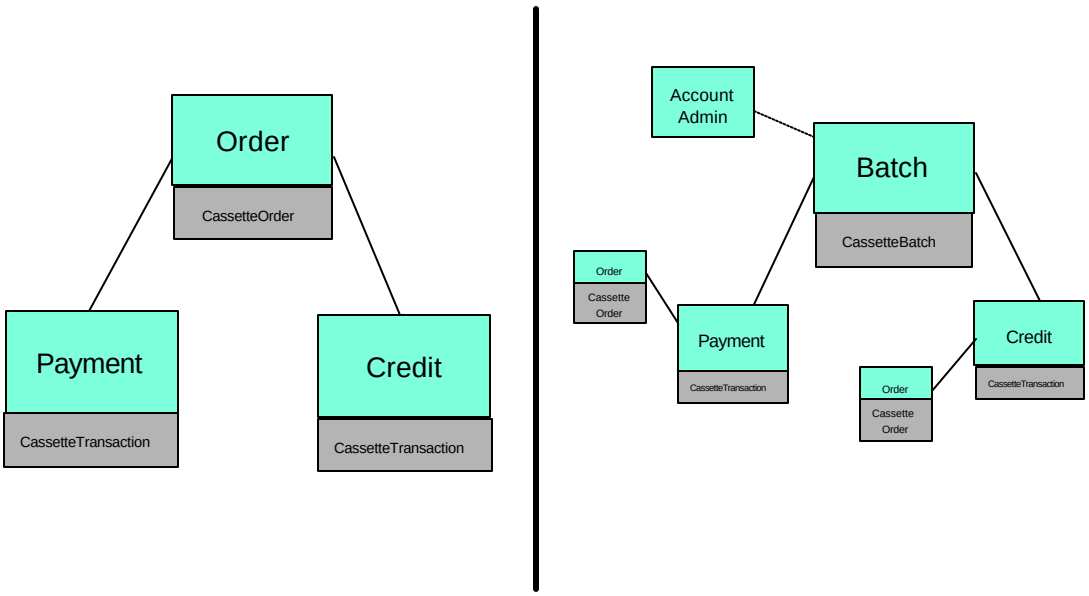
Administration Model



Administration Model *(continued)*

- PayServer - represents the Payment Manager itself - only 1 of these per Payment Manager
- CassetteAdmin - describes a cassette - 1 per installed cassette
- MerchantAdmin - describes a merchant - 1 per merchant
- PaySystem - describes an association between 1 merchant and 1 cassette (UI calls this the "Merchant Cassette Settings")
- AccountAdmin - describes the relationship between a merchant and a finiancial institution under a given cassette - *n* per PaySystem
- Cassette-specific objects - defined by a cassette to describe a protocol-specific administration object
  - SystemCassetteObject - cassette scope
  - MerchantCassetteObject - merchant scope (Brand is an example)

Financial Model



Financial Model *(continued)*

- Order - represents a buyer's agreement to pay *using a specific payment instrument* (credit card, check, etc.)
- Payment - one transfer of funds from the buyer to the seller using the payment instrument identified by the owning Order. May be 0 or more of these per Order
- Credit - one refund of funds from the seller to the buyer using the payment instrument identified by the owning Order. May be 0 or more of these per Order
- Batch - a set of Payments and Credits to be settled as a group with the financial institution represented by the associated Account. These Payments and Credits typically belong to different Orders.

The Application Programming Interface

The framework defines these commands:

- Financial
  - ReceivePayment
  - AcceptPayment
  - CloseOrder
  - CancelOrder
  - Approve
  - ApproveReversal
  - Deposit
  - DepositReversal
  - Refund
  - RefundReversal
  - BatchOpen
  - BatchClose
  - BatchPurge
  - DeleteBatch
- Administration
  - CreateAccount
  - ModifyAccount
  - DeleteAccount
  - Others
- Queries
  - XML formats

Memory Management for Financial Objects

- The framework guarantees a single in-memory copy of any given financial object through its Order and Batch caches.
- Order and Batch objects contain references to associated Payment, Credits, CassetteOrder and CassetteBatch objects.
- Only the framework constructs framework objects.
- The framework will ask the cassette to create or resurrect cassette objects as needed. Your cassette is responsible for constructing and resurrecting cassette objects upon request.
- Cassettes must adhere to the following rules to ensure the integrity and effectiveness of the Order and Batch caches:
  - do not save references to financial objects across requests
  - do not pass references to these objects to another thread, including service threads
  - do not keep your own cache or lists of in-memory objects
  - do not make "copies" of any of these in-memory objects

Memory Management for Administration Objects

- All administration objects are kept in memory full-time
- No direct references from Framework administration objects to cassette extensions - cassette is responsible for this association.
- Cassettes may maintain their own collections which reference their administration objects
- Cassette extensions should implement the Archivable interface to allow for storage to the database

Synchronization

- Object synchronization
  - The Framework provides all synchronization necessary to ensure data integrity and thread safety while allowing for as much parallelism as possible. *Cassettes should not have to perform any synchronization for safe PM object access.*
  - The Framework decides which locks to obtain based upon the request type.
  - A hierarchical locking scheme ensures that multiple readers can have access to objects but writers must be single-threaded.
- Database serialization (ETillArchive lock)- exists for historical reasons.

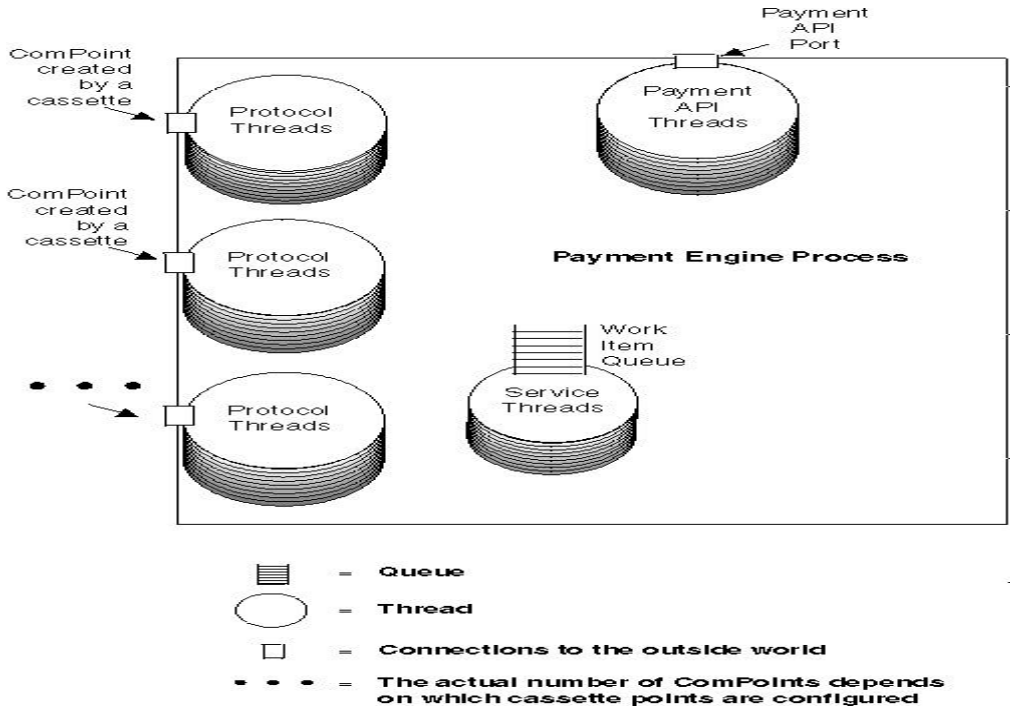
Parameter Validation and Conversion

- Framework parses and syntax checks the cassette-specific API parameters based on parameter validation rules supplied by the cassette
- Converts from byte-array to the specified Java formats
- Does not provide:
  - Existence checks
  - Context checks
  - Semantic checks
  - Consistency checks
- Casettes can extend ParameterValidationItem (for example, card account number validation)

Thread Management

- The framework provides all threads and controls the scheduling of cassette processing. Several Framework services are dependent upon these threads:
  - Database commit points
  - Object synchronization
  - Cache management
- Cassettes *should not create* their own threads
- Cassette software should be written in a way that any function can run at any time in any thread (thread safe).

Payment Engine thread model



Database Access

- ETillArchive final class manages access to the database using JDBC calls.
  - used to store and retrieve persistent information
  - used to resurrect objects
  - used to read the configuration
  - used to protect the cassette from platform and database manager inconsistencies
- CommitPoint object (one per Payment Manager thread) is used to ensure that all associated Framework and cassette database updates occur as a single unit of work.
- Archivable interface defines methods needed to save an object to the database under control of a CommitPoint using the ETillArchive class.

Access Control

- Restricted access to function
- Managed completely within the Payment Servlet
- Cassettes never have to worry about this
- Only authorized users can view or update merchant data
- Merchant can restrict users to specific roles:
  - Clerk
  - Supervisor
  - Merchant Administrator
- Payment Manager Administrator authority used for managing multiple merchants

Scheduling Background and Timed Tasks

- Not used in our LDBCard cassette, but is illustrated in KitCash cassette. Useful for asynchronous retries and long running processes.
- Framework maintains a pool of service threads to service a CassetteWorkItem queue.
- Framework maintains a timer thread to service a Timeable CassetteWorkItem queue.
- Used by cassettes to schedule future work and to offload work from the running thread.
- CassetteWorkItems are cassette-generated objects
- Since CassetteWorkItem construction is based upon a particular xxxRequest object, the Framework ensures that the correct object locks are obtained before passing the work item to the cassette for processing.

Protocol Message Management

- Not used in LDBCard cassette, but is illustrated by the KitCash cassette.
- Used for "server-side" processing of new *inbound* protocol messages, typically from a *Wallet or browser plug-in. NOT* for "client-side" messages.
- ComPoint and ETillconnection interfaces can be used to create cassette classes to exchange protocol messages with "outside world".
- Framework provides threads dedicated to listening for these incoming messages.
- ComPoint represents the "listening point". It waits for notification of an incoming message and return an EtillConnection object.
- The ETillConnection object is passed to the cassette responsible for handling the incoming message.
- The ComPoint object waits for the next incoming message.

Event Notification

- Asynchronous Notification of significant events to registered event listener applications
- Main event of interest is the object state change.
- Cassette manages object state. No other cassette action necessary
- The framework notifies external business systems listening for the state changes
- Cassettes can provide their own events if necessary

Exception Management

- A cassette is required to catch each Exception and each routine abnormal condition
- If a cassette encounters an unrecoverable error, it must throw a framework exception:
  - ETillAbortOperation : terminates the current operation
    - Merchant application gets return codes
    - Used for parameter errors
    - Used for usage errors
    - Used for execution failures
  - ETillCassetteException : stops the cassette
    - A subclass of RuntimeException
    - Used for assertion failures

Tracing Internal Events

- Granular trace facility. Some examples are
  - Function Entry and Exit
  - Database Read, Write and Commit
  - Error occurred
  - Debug
- Used during development and test
- Validation that the cassette is operating correctly
- Assistance with debugging efforts
- Trace is usually turned off for production
  - Use error log for production errors and runtime information
  - In production, trace facility used as a last resort

Tracing Internal Events *(continued)*

Sample Trace call:

```
if (Trace.isAnyoneTracing()) {
   Trace.traceFunctionEntry(
      TRACE_ID,
      "LdbCardAccount.connect()"
   );
}
```

Logging Errors

- The framework provides a logging facility for
  - Informational messages
  - Error messages
- Cassette dependent properties file used for messages facilitates easy translation
- Cassettes should capture critical data when errors are detected or when key events occur in order to reduce service costs and problem identification effort
- Should not be used for
  - routine or expected conditions that occur frequently
  - reporting syntax errors on inbound API commands (return codes notify the offending application of the errors)

Logging Errors *(continued)*

Sample Error Log:

```
ErrorLog.logError(
   RB_ID,
   MSG_ACCOUNT_ICV_FAILURE,
   varException,
   this.getMerchant(),
   this.getAccountNumber(),
   this.getUser(),
   this.getObscurePassword(),
   this.getUrl()
);
```