PROPERTY MANAGEMENT APPLICATION

By Marthand Bhargav

Phase 7: Integration & External Access - Documentation

1. Introduction

Purpose of Integration & External Access

Integration is a critical component of modern enterprise applications, enabling Salesforce to communicate with external systems, third-party services, and other platforms. Phase 7 focuses on establishing secure, reliable connections between our Rental Property Management system and external services, with particular emphasis on payment processing through Authorize.Net.

In today's interconnected business environment, no system operates in isolation. Our implementation demonstrates real-world integration patterns that enable:

- Secure payment processing through external gateways
- Real-time data synchronization across platforms
- Event-driven architectures for scalability
- API-based communication following industry standards
- Compliance with security and authentication best practices

Importance of Integration Components

- **Named Credentials**: Centralized, secure storage of authentication details eliminating hardcoded credentials
- **External Services**: Declarative integration with REST APIs without custom code
- **Web Services**: Programmatic access via REST and SOAP for complex integrations
- **Callouts**: Synchronous communication with external systems from Apex
- **Platform Events**: Event-driven architecture for loosely coupled integrations
- **Change Data Capture**: Real-time data replication and synchronization
- **Salesforce Connect**: Virtual integration accessing external data without storing in Salesforce
- **API Limits**: Understanding and managing governor limits for scalable solutions
- **OAuth & Authentication**: Industry-standard security protocols for authorized access
- **Remote Site Settings**: Security configuration for outbound HTTP requests

2. Authorize.Net Payment Gateway Integration

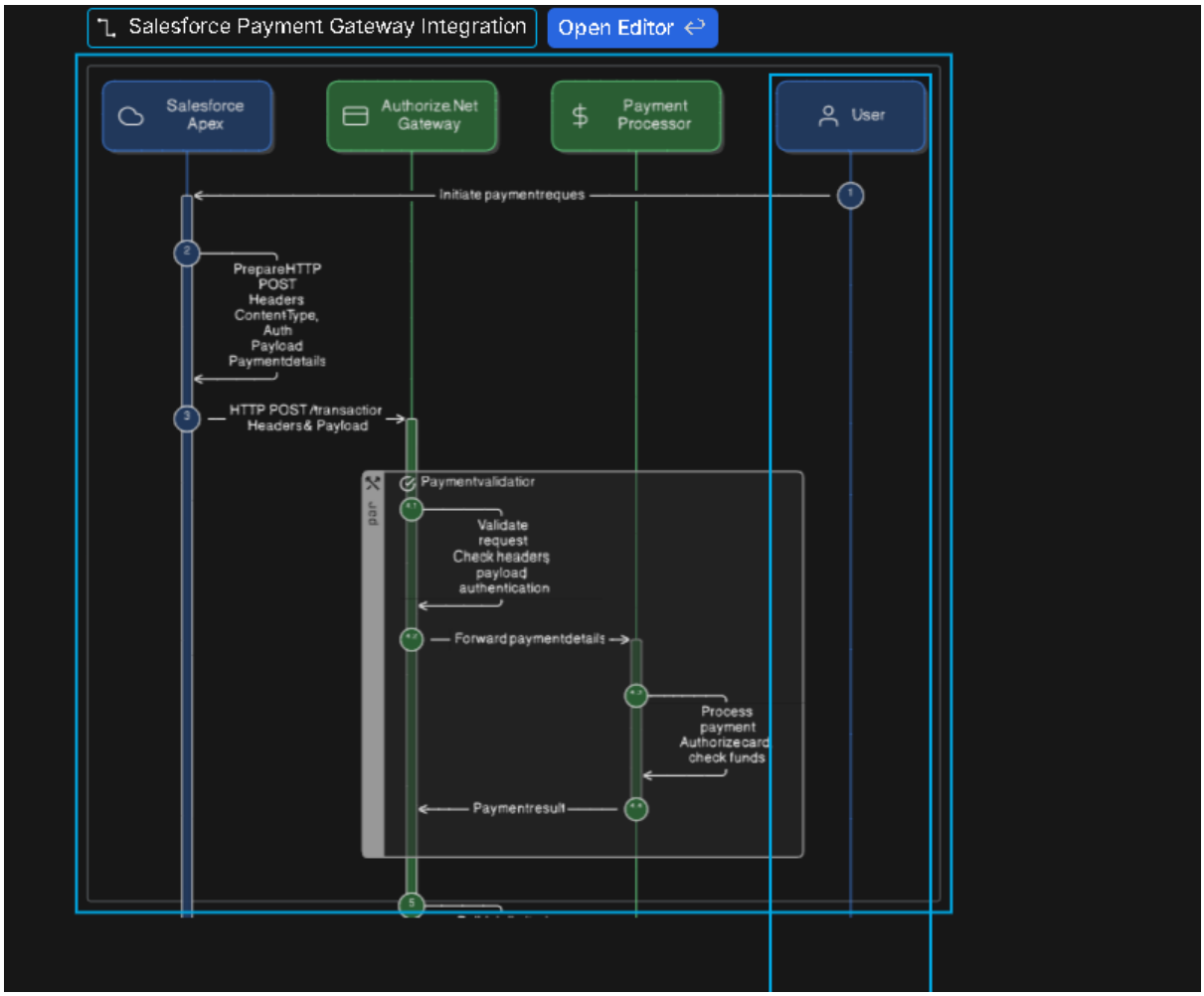
2.1 Integration Overview

Our primary external integration connects the Rental Property Management system with Authorize.Net, a leading payment gateway provider. This integration enables secure credit card processing for rental payments directly within Salesforce.

Business Requirements Addressed:

- Accept credit card payments from renters
- Secure handling of sensitive payment data (PCI compliance)
- Real-time transaction processing and authorization
- Automatic payment record creation with transaction details
- Transaction tracking and reconciliation

Integration Architecture:



2. Remote Site Settings Configuration

Before Salesforce can make outbound HTTP requests to Authorize.Net, we must configure Remote Site Settings to whitelist the external endpoint.

Configuration Steps:

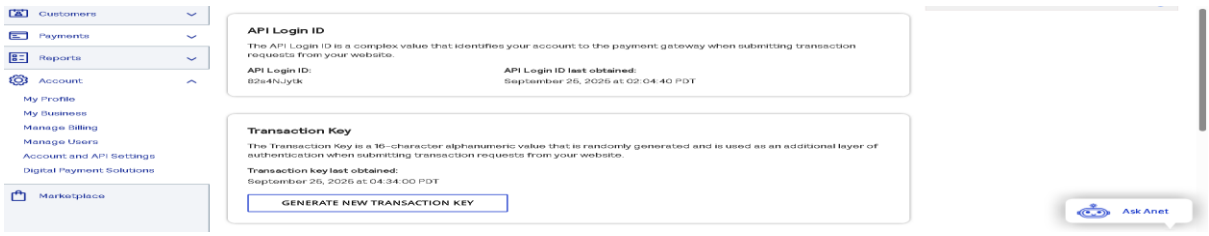
1. **Navigate to Remote Site Settings**:
    - Setup → Security → Remote Site Settings
    - Click "New Remote Site"
2. **Configure Authorize.Net Endpoints**: **Test Environment**:
    - **Remote Site Name**: AuthorizeNet_Test
    - **Remote Site URL**: https://test.authorize.net
    - **Disable Protocol Security**: Unchecked (enforce HTTPS)
    - **Description**: "Authorize.Net Test Gateway for Payment Processing"
    - **Active**: Checked

Production Environment:

- **Remote Site Name**: AuthorizeNet_Production
- **Remote Site URL**: https://secure.authorize.net
- **Disable Protocol Security**: Unchecked
- **Description**: "Authorize.Net Production Gateway for Live Payments"
- **Active**: Checked (only when ready for production)

Security Considerations:

- Always use HTTPS endpoints to encrypt data in transit
- Separate test and production configurations
- Document the purpose of each remote site
- Review remote sites periodically for unused entries



2.3 Custom Settings for Credential Management

Rather than hardcoding API credentials, we use Custom Settings to store Authorize.Net authentication details securely.

Custom Setting Configuration:

Object Name: Authorize_net_Setting__c

Setting Type: Hierarchy (allows override at profile, user, or org level)

Fields:

- Name (Standard): Unique identifier (e.g., "API Login", "TransKey")
- Value__c (Custom Text): Encrypted field storing credential values

Records Created:

1. **API Login Credential**:

Name: API Login

Value__c: [Your Authorize.Net API Login ID]

2. **Transaction Key Credential**:

Name: TransKey

Value__c: [Your Authorize.Net Transaction Key]

Advantages of Custom Settings:

- Credentials stored separately from code
- Can be updated without code deployment
- Different values per environment (Sandbox vs Production)
- Access controlled via permissions
- Encrypted at rest when using platform encryption

Best Practice Implementation:

```
public static void getAuthNetCreds(){
    // Retrieve credentials from Custom Settings
    Authorize_net_Setting__c apiloginsetting =
        Authorize_net_Setting__c.getInstance('API Login');
    Authorize_net_Setting__c apitranskeysetting =
        Authorize_net_Setting__c.getInstance('TransKey');

    APILOGIN = apiloginsetting.value__c;
    APITRANSKEY = apitranskeysetting.value__c;
}
```

2.4 Named Credentials (Best Practice Alternative)

While our current implementation uses Custom Settings, **Named Credentials** represent a more modern and secure approach for storing external authentication details.

Why Named Credentials Are Superior:

- Credentials never exposed to Apex code or logs
- Automatic credential injection in HTTP callouts
- Support for OAuth 2.0, JWT, and other protocols
- Certificate-based authentication
- Per-user vs. per-org authentication models
- Simplified endpoint management

Migration Path to Named Credentials:

Step 1: Create Named Credential:

Setup → Named Credentials → New Named Credential

Configuration:

- **Label**: Authorize.Net Payment Gateway
- **Name**: AuthorizeNet_Gateway
- **URL**: https://test.authorize.net/gateway/transact.dll
- **Identity Type**: Named Principal

- **Authentication Protocol**: Password Authentication
- **Username**: [Your API Login ID]
- **Password**: [Your Transaction Key]
- **Generate Authorization Header**: Checked
- **Allow Merge Fields in HTTP Header**: Checked
- **Allow Merge Fields in HTTP Body**: Checked

Step 2: Update Apex Callout:

```
// Before (with Custom Settings)
req.setEndpoint('https://test.authorize.net/gateway/transact.dll');
req.setHeader('Authorization', 'Basic ' + encodedCreds);

// After (with Named Credentials)
req.setEndpoint('callout:AuthorizeNet_Gateway');
// Authorization header automatically added by Salesforce
```

Benefits Realized:

- No credential retrieval logic in Apex
- Credentials not visible in debug logs
- Easier credential rotation without code changes
- Centralized authentication management

---

3. Web Services Implementation

3.1 HTTP Callout to Authorize.Net (REST-style)

Our integration uses a REST-style HTTP POST request to communicate with Authorize.Net's AIM (Advanced Integration Method) API.

API Endpoint Details:

- **Test URL**: https://test.authorize.net/gateway/transact.dll
- **Production URL**: https://secure.authorize.net/gateway/transact.dll
- **Method**: POST
- **Content-Type**: application/x-www-form-urlencoded
- **Response Format**: Delimited text (customizable delimiter)

3.2 Request Construction

Complete Implementation:

```
public static authnetresp_wrapper authdotnetCharge(authnetreq_wrapper input) {
    // Retrieve credentials from Custom Settings
    getAuthNetCreds();

    // Construct HTTP request
    HttpRequest req = new HttpRequest();
    req.setEndpoint('https://test.authorize.net/gateway/transact.dll');
    req.setMethod('POST');
    req.setTimeout(120000); // 120 second timeout

    // Build message payload
    Map<String, String> messagestring = new Map<String, String>();

    // Authentication Fields
    messagestring.put('x_login', APILOGIN);
    messagestring.put('x_tran_key', APITRANSKEY);
```

```
messagestring.put('x_version', '3.1');
messagestring.put('x_delim_data', 'TRUE');
messagestring.put('x_delim_char', ';'); // Semicolon delimiter
messagestring.put('x_relay_response', 'FALSE');

// Transaction Type
messagestring.put('x_type', 'AUTH_CAPTURE'); // Authorize and capture in on
messagestring.put('x_method', 'CC'); // Credit Card

// Payment Information
messagestring.put('x_card_num', input.ccnum);
messagestring.put('x_exp_date', input.ccexp); // Format: MMYY
messagestring.put('x_card_code', input.ccsec); // CVV
messagestring.put('x_amount', input.amt);
messagestring.put('x_description', 'Your Transaction: ' + input.ordername);

// Billing Information (AVS verification)
messagestring.put('x_first_name', input.firstname);
messagestring.put('x_last_name', input.lastname);
messagestring.put('x_address', input.billstreet);
```

```
String encodedmsg = '';
for (String s : messagestring.keySet()) {
    String v = messagestring.get(s);
    if (String.isBlank(v)) v = ''; // Handle null values
    encodedmsg += s + '=' + EncodingUtil.urlEncode(v, 'UTF-8') + '&';
    System.debug('TRACE: message bit ' + s + ' added');
}
encodedmsg += 'endofdata';

System.debug('TRACE: Encoded Message: \n\n' + encodedmsg);
req.setBody(encodedmsg);

// Send request and capture response
Http http = new Http();
String resp = http.send(req).getBody();
System.debug('TRACE: Response: ' + resp);

// Parse delimited response
List<String> responses = resp.split(';');
authnetresp_wrapper parsedResponse = parseIntoResponseWrapper(responses)
```

Key Implementation Details:

1. **Timeout Configuration**: Set to 120 seconds to allow for network latency and gateway processing time

2. **URL Encoding**: Critical for special characters in billing information

apex

```
EncodingUtil.urlEncode(v, 'UTF-8')
```

3. **Null Handling**: Prevents errors when optional fields are blank

apex

```
if (String.isBlank(v)) v = '';
```

4. **Debug Logging**: Comprehensive logging for troubleshooting (remove in production)

3.3 Request Wrapper Class

The authnetReq_Wrapper class encapsulates all data needed for a payment transaction:

```
public class authnetReq_Wrapper {
    // Order Information
    public String ordername {get; set;}

    // Credit Card Information
    public String ccnum {get; set;}
    public String ccexp {get; set;} // Format: MMYY
    public String ccsec {get; set;} // CVV/CVC

    // Transaction Amount
    public String amt {get; set;}

    // Billing Information
    public String firstname {get; set;}
    public String lastname {get; set;}
    public String billstreet {get; set;}
    public String billcity {get; set;}
    public String billstate {get; set;}
    public String billzip {get; set;}
```

```
    // Billing Information
    public String firstname {get; set;}
    public String lastname {get; set;}
    public String billstreet {get; set;}
    public String billcity {get; set;}
    public String billstate {get; set;}
    public String billzip {get; set;}

    // Bank Account Information (for eCheck - future enhancement)
    public String transid {get; set;}
    public String routingnumber {get; set;}
    public String accountnumber {get; set;}
    public String bankaccounttype {get; set;}
    public String bankname {get; set;}
    public String bankaccountname {get; set;}

    public authnetreq_wrapper() {}
```

Usage in Extension Controller:

```
// Create request wrapper
api_AuthorizeDotNet.authNetReq_Wrapper req =
    new api_AuthorizeDotNet.authNetReq_Wrapper();

// Populate from Payment record
req.amt = String.valueOf(thisPayment.Amount__c);
req.firstname = thisPayment.Billing_Name__c.substringBefore(' ');
req.lastname = thisPayment.Billing_Name__c.substringAfter(' ');
req.ccnum = thisPayment.Credit_Card_Number__c;
req.ccexp = monthMap.get(thisPayment.Credit_Card_Expiration_Month__c) +
            thisPayment.Expiration_Year__c;
req.ccsec = thisPayment.Credit_Card_Security_Card__c;

// Process payment
api_AuthorizeDotNet.authNetResp_Wrapper res =
    api_AuthorizeDotNet.authdotnetCharge(req);
```

3.4 Response Parsing

Authorize.Net returns a delimited text response containing 45+ fields. Our parser extracts these into a structured wrapper class.

Response Format Example:

1;1;1;This transaction has been approved.;ABC123;Y;123456789;INV001;Payment;100.00;CC;auth_capture;...

Parser Implementation:

Response Code Interpretation:

- **1 (Approved)**: Transaction successful, funds captured
- **2 (Declined)**: Card declined by issuing bank
- **3 (Error)**: System error or invalid data
- **4 (Held for Review)**: Fraud detection triggered

3.5 Error Handling and Transaction Validation

Implementation in Extension Contro

```
public static authNetResp_Wrapper parseIntoResponseWrapper(List<String> input) {
    authNetResp_Wrapper temp = new authNetResp_Wrapper();

    // Core Transaction Fields
    temp.responseCode = input[0];           // 1=Approved, 2=Declined, 3=Error
    temp.ResponseSubcode = input[1];
    temp.ResponseReasonCode = input[2];
    temp.ResponseReasonText = input[3];     // Human-readable message
    temp.AuthorizationCode = input[4];      // Bank authorization code
    temp.AVSResponse = input[5];            // Address verification
    temp.TransactionID = input[6];          // Unique transaction identifier

    // Invoice and Description
    temp.InvoiceNumber = input[7];
    temp.Description = input[8];
    temp.Amount = input[9];
    temp.Method = input[10];
    temp.TransactionType = input[11];
```

ller:

```
  // Customer Information
  temp.CustomerID = input[12];
  temp.FirstName = input[13];
  temp.LastName = input[14];
  temp.Company = input[15];

  // Billing Address
  temp.Address = input[16];
  temp.City = input[17];
  temp.State = input[18];
  temp.ZIPCode = input[19];
  temp.Country = input[20];
  temp.Phone = input[21];
  temp.Fax = input[22];
  temp.EmailAddress = input[23];
```

```
temp.ShipToFirstName = input[24];
temp.ShipToLastName = input[25];
temp.ShipToCompany = input[26];
temp.ShipToAddress = input[27];
temp.ShipToCity = input[28];
temp.ShipToState = input[29];
temp.ShipToZIPCode = input[30];
temp.ShipToCountry = input[31];

// Additional Transaction Data
temp.Tax = input[32];
temp.Duty = input[33];
temp.Freight = input[34];
temp.TaxExempt = input[35];
temp.PurchaseOrderNumber = input[36];
temp.MD5Hash = input[37];                // Security hash
temp.CardCodeResponse = input[38];       // CVV verification
temp.CardholderAuthenticationVerificationResponse = input[39];
```

```
// Card Information (masked)
temp.AccountNumber = input[40];          // Last 4 digits
temp.CardType = input[41];               // Visa, MasterCard, etc.

// Split Tender (partial payments)
temp.SplitTenderID = input[42];
temp.RequestedAmount = input[43];
temp.BalanceOnCard = input[44];

return temp;
```

Common Error Scenarios:

1. **Declined Card** (Code 2):
    - Insufficient funds
    - Expired card
    - Invalid card number
    - Card restricted by issuer
2. **Validation Errors** (Code 3):
    - Missing required fields
    - Invalid expiration date format
    - AVS mismatch
    - Invalid CVV
3. **Gateway Errors**:
    - Invalid API credentials
    - Timeout (network issues)
    - Gateway maintenance

---

4. API Limits and Governor Limits

4.1 Salesforce Governor Limits for Callouts

Understanding and managing governor limits is critical for scalable integrations.

Key Limits:

Limit Type	Synchronous	Asynchronous	Notes
Total HTTP Callouts	100	100	Per transaction
Callout Timeout	120 seconds	120 seconds	Maximum
Maximum Response Size	6 MB	6 MB	Uncompressed
Maximum Request Size	6 MB	6 MB	Including headers
Total Heap Size	6 MB	12 MB	All memory usage

4.2 Optimization Strategies

1. Batch Processing for Multiple Payments:

Instead of individual callouts for each payment:

```
for (Payment__c payment : payments) {
    processPayment(payment); // Each makes a callout
}

// ✅ Good: Batch via future/queueable
@future(callout=true)
public static void processPaymentsBatch(List<Id> paymentIds) {
    List<Payment__c> payments = [SELECT ... FROM Payment__c WHERE Id IN :paymen
    for (Payment__c payment : payments) {
        // Process with callout
        // Stays within 100 callout limit per future execution
    }
}
```

2. Asynchronous Processing:

```
@future(callout=true)
public static void processPaymentAsync(Id paymentId) {
    // Callout logic here
    // Benefits: Increased timeout, separate governor limits
}
```

3. Caching Strategy:

```
// Cache authentication tokens if API supports it
private static Map<String, String> authTokenCache = new Map<String, String>();

public static String getAuthToken() {
    if (authTokenCache.containsKey('token')) {
        return authTokenCache.get('token');
    }
    // Make callout to get token
    String token = fetchNewToken();
    authTokenCache.put('token', token);
    return token;
}
```

4.3 Authorize.Net API Limits

Transaction Limits:

- **Test Environment**: Unlimited transactions
- **Production**: Based on merchant account agreement
- **Rate Limiting**: ~10 requests per second recommended
- **Duplicate Transaction Window**: 2 minutes (prevents accidental duplicate charges)

Best Practices:

- Implement idempotency checks before callouts
- Store transaction IDs to prevent duplicate processing
- Use test mode for development/UAT
- Monitor transaction volumes via Authorize.Net dashboard

---

5. Authentication and Security

5.1 Authorize.Net Authentication Method

Our implementation uses **API Login ID and Transaction Key** authentication (basic authentication variant).

Authentication Flow:

1. Retrieve credentials from Custom Settings/Named Credentials

2. Include in request body as parameters:

- x_login: API Login ID

- x_tran_key: Transaction Key

3. Authorize.Net validates credentials

4. Processes transaction if valid

Security Measures:

- Credentials stored in Custom Settings (encrypted at rest)
- Never logged or exposed in error messages
- Transmitted over HTTPS (TLS 1.2+)
- Separate test and production credentials

5.2 PCI Compliance Considerations

Our Implementation Strategy:

✅ **PCI-Compliant Practices**:

1. **No Storage of Sensitive Data**:
    - Credit card numbers NOT stored in Salesforce
    - CVV codes NOT stored (input only, never persisted)
    - Only store last 4 digits (from Authorize.Net response)
2. **Secure Transmission**:
    - All data transmitted via HTTPS
    - TLS 1.2 or higher enforced
    - Remote Site Settings restrict to secure endpoints
3. **Minimal Data Handling**:
    - Payment form submits directly to Apex
    - Data held in memory only during transaction
    - No logging of full card numbers

Payment Record Fields (Safe to Store):

```
thisPayment.Authorize_net_Transaction_ID__c = res.TransactionID;
thisPayment.Authorize_net_Authorization_Code__c = res.AuthorizationCode;
thisPayment.Card_Type__c = res.CardType; // Visa, MC, etc.
thisPayment.Last_Four_Digits__c = res.AccountNumber; // XXXX
```

5.3 OAuth 2.0 (Future Enhancement)

While our current integration uses API key authentication, Authorize.Net and other modern APIs support OAuth 2.0.

OAuth 2.0 Advantages:

- Token-based authentication (more secure)
- Tokens expire and refresh automatically
- Granular permission scopes
- Easier credential rotation

Implementation with Named Credentials:

Authentication Protocol: OAuth 2.0

Grant Type: Client Credentials

Token Endpoint: https://api.authorize.net/oauth/token

Client ID: [Your Client ID]

Client Secret: [Your Client Secret]

Scope: transaction:process, transaction:read

Apex Usage (Simplified with Named Credentials):

```
HttpRequest req = new HttpRequest();
req.setEndpoint('callout:AuthorizeNet_OAuth/transactions');
// OAuth token automatically injected by Salesforce
req.setMethod('POST');
req.setBody(jsonPayload);
```

6. Platform Events (Event-Driven Architecture)

6.1 Use Case: Payment Processing Events

Platform Events enable real-time, event-driven communication within Salesforce and to external systems.

Scenario: Broadcast payment events for downstream processing (accounting, reporting, notifications).

Platform Event Definition:

Object Name: Payment_Processed__e

Fields:

- Payment_Id__c (Text): Salesforce Payment record ID
- Transaction_ID__c (Text): Authorize.Net transaction ID
- Amount__c (Currency): Payment amount
- Status__c (Picklist): Success, Failed, Pending
- Timestamp__c (DateTime): Processing time
- Renter_Email__c (Email): For notifications

6.2 Publishing Events

Apex Implementation:

```
public static void publishPaymentEvent(Payment__c payment, String status) {
    // Create platform event
    Payment_Processed__e event = new Payment_Processed__e();
    event.Payment_Id__c = payment.Id;
    event.Transaction_ID__c = payment.Authorize_net_Transaction_ID__c;
    event.Amount__c = payment.Amount__c;
    event.Status__c = status;
    event.Timestamp__c = System.now();
    event.Renter_Email__c = getRenterEmail(payment.Statement__c);

    // Publish event
    Database.SaveResult sr = EventBus.publish(event);

    if (!sr.isSuccess()) {
        for (Database.Error err : sr.getErrors()) {
            System.debug('Error publishing event: ' + err.getMessage());
        }
    }
}
```

Integration in Payment Flow:

```
thisPayment.Status__c = 'Paid';
upsert thisPayment;

// Publish event for downstream processing
publishPaymentEvent(thisPayment, 'Success');
```

6.3 Subscribing to Events

Trigger Subscription:

```
trigger PaymentProcessedEventTrigger on Payment_Processed__e (after insert) {
    List<Id> successfulPaymentIds = new List<Id>();

    for (Payment_Processed__e event : Trigger.new) {
        if (event.Status__c == 'Success') {
            successfulPaymentIds.add(event.Payment_Id__c);
        }
    }

    if (!successfulPaymentIds.isEmpty()) {
        // Trigger accounting integration
        AccountingIntegrationService.syncPayments(successfulPaymentIds);

        // Send email notifications
        NotificationService.sendPaymentConfirmations(successfulPaymentIds);
    }
}
```

Flow Subscription:

- Create a Flow triggered by Platform Event
- Add criteria: Status__c equals Success
- Actions: Update related records, send email, create tasks

6.4 Benefits of Platform Events

Decoupling:

- Payment processing independent of notification logic
- Easier to add new subscribers without modifying payment code

Scalability:

- Asynchronous processing
- Events published immediately, processed later
- Handles high-volume scenarios

Reliability:

- Events stored for 72 hours (High Volume) or 24 hours (standard)
- Replay functionality for failed subscribers
- Guaranteed delivery