Discover Payment Services

Common Payments Platform (CPP)
Software Architecture Document

Version 0.12

Revision History

Date	Version	Description	Author
04/JAN/2016	0.7	• Reformatted and removed a lot of the detailed information in order to simplify the content and provide more focused content on the architecturally-significant use-cases	Mike Knauff
08/JAN/2016	0.8	• Updated the following Use-Cases with feedback from the PSA Team o Fix Daily Financial Event Recon Errors o Reconcile Financial Events o Route Financial Event • Updated the Logical View for Architecture Components with additional event reconciliation components • Updated the Context View with additional event reconciliation components	Mike Knauff
08/JAN/2016	0.9	• Updated the Invoke External Application Request Use-Case • Updated the Invoke External Service Request Use-Case • Updated the Invoke Internal Service Request Use-Case • Completed the Component Catalog for the Context View • Updated the Architecture Components Diagram • Updated the Context Diagram	Mike Knauff
20/JAN/2016	0.10	• Added the *Execute Logical Transaction* use-case • Added the *Rollback/Cancel Logical Transaction* use-case • Replaced references to *Financial* Events with *Critical* Events in order to make the use case realizations more generic and re-usable	Mike Knauff

Date	Version	Description	Author
03/FEB/2016	0.11	• Enhanced & updated the Context View Diagram and descriptions • Updated Sequence Diagrams based upon context view updates • Added Upgrade Micro-Service Minor Version Use-Case	Mike Knauff
04/APR/2016	0.12	• Revised major use-case diagrams with consistent naming • Added Data Flow Diagram	Mike Knauff

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

This document provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made on the system.

1.2 Scope

The Common Payments Platform (CPP)Software Architecture Documents provides the high-level overview of the architecturally-significant aspects of the platform. The CPP is a large and complex system and this document describes each of the major components and how they are integrated to realize the qualities or non-functional requirements (NFRs) of the system.

The details of each of the components are provided by the individual Software Architecture Documents for each of the individual components.

1.3 Definitions, Acronyms, and Abbreviations

1.3.1 ACID Properties

ACID Properties are those properties that are applied to updates to data, which are maintained by transactions and that include atomic, consistent, isolated, and durable.

1.3.2 Black Box Testing

Black Box Testing is the testing of software components purely through observing its external behavior by controlling inputs and observing outputs with no knowledge of internal paths, structures, or implementation of the software being tested.

1.3.3 Critical Event

A *Critical* Event is any (asynchronous) event that occurs on the CPP platform that needs to be guaranteed that it was delivered and processed by subscribers. The most common *Critical Events* are financial events that need affect settlement with Tenants and Tenant Customers. There are other *Critical Events* that are part of distributed logical transactions that must be guaranteed to take place in order to ensure that the platform is in a consistent state.

1.3.4 ETL

ETL is the acronym for *Extract, Transform,* and *Load*. *ETL* processes are utilized by the CPP Data Services to transform data into the proper formats as it is moved from or to data batch biles.

1.3.5 ESXi

ESXi is the acronym for VMware's *Elastic Sky X Integrated* hypervisor software product.

1.3.6 Idempotent

Idempotent refers to the property of a service or operation which leaves the resource that is operated on in the same state no matter how many times and identical operation is called upon the resource. It is key for operations to be *Idempotent* within architectures that feature asynchronous messaging or synchronous messaging that does not provide distributed transaction support.

1.3.7 Infrastructure as a Service (IaaS)

Infrastructure as a Service or *IaaS* is the delivery of computer hardware (servers, networking technology, storage, and data center space) as a service through the use of virtualized components that run on top of their physical equivalents. The use of virtualized server, networking, and storage components allows the deployment of these components "on demand" without the need for lengthy "request, procure, deploy, and configure" timeframes. Also the use of virtualized components allow for more fault-tolerant configurations of the virtualize hardware components over their bare-metal equivalents.

1.3.8 Multi-Tennant

Multi-Tennant refers to the ability of the platform and the data model to accommodate multiple payment brands (e.g., Diners Club International, Discover Network, PULSE Network, PayPal, Ariba, etc.) and their unique products, processing rules, customers, customer agreements, etc. within the same structure and within the same application or service instance.

1.3.9 PCF Space

A *PCF Space* corresponds to a lifecycle stage (development, development integration, QA, performance testing, production, etc.).

1.3.10 PCI-DSS

PCI-DSS is the acronym for the *Payment Card Industry Data Security Standard*, which is the set of security requirements that entities that handle payment card transactions and data, and that must conform to in their processing environments.

1.3.11 Platform

The *Platform* is the group of applications, services, and data repositories that comprise the Common Payments Platform as whole, and that is responsible for providing payment products in a shared, multi-tenant environment.

1.3.12 Platform as a Service (PaaS)

Platform as a Service or *PaaS* is the delivery of a solution stack, which is an integrated set of software that provides everything a developer needs to build an application for both software development and runtime. The use of PaaS allows for "on demand" deployment of software services and applications. Also the use of virtualized application containers allow for more fault-tolerant configurations applications and application services. *PaaS* runs on top of IaaS.

1.3.13 White Box Testing

White Box Testing is the testing of software components by evaluating internal logic paths, code structures, and implementation of the software and controlling the component inputs based with the intent of exercising the internal code structure to evaluate its correctness.

1.4 References

- Common Payments Platform Reference Architecture Document v2.1
- Common Payments Platform Architecture Strategy v3.1

1.5 Overview

*[This subsection describes what the rest of the **Software Architecture Document** contains and explains how the **Software Architecture Document** is organized.]*

2. Architectural Representation

2.1 Architecture Goals

2.1.1 Provide multi-tenant support for DFS payment brands and white-label tenants

2.1.2 Provide continuously available real-time and near real-time services

2.1.3 Provide for horizontal scalability

2.1.4 Provide for adaptability in the following dimensions

2.1.4.1 Add, update, remove service components with minimum effects

- Independent service components

2.1.4.2 Add, update, change technologies with minimum effects

- Independent layers in the technology stack

2.1.4.3 Acquire new businesses, business models, and partner relationships

- Adaptable (Meta) domain model

2.1.4.4 Integration of a variety of service providers (DFS and vendor) and expose as a single platform

- Uniform and industry standard APIs and communication protocols

2.1.5 Facilitate continuous integration and delivery

2.2 Architecture Strategies

2.2.1 Canonical Formats

Provides for modifiability, adaptability, extensibility, reusability

2.2.1.1 Shared internal message and data formats

2.2.2 Independent Components

Provides for reusability, modifiability, adaptability

2.2.2.1 Service Directory (Real-Time Component Integration)

2.2.2.2 Event/Message Bus (Near Real-Time Component Integration)

2.2.2.3 File Transfer Manager (Offline Component Integration)

2.2.3 Layers Pattern

Provides for reusability, portability, modifiability, adaptability

2.2.3.1 Virtual machines

2.2.3.2 Adapter or wrapper components

2.2.3.3 APIs

2.2.4 Meta-Model

Provide for adaptability, modifiability, reusability

2.2.4.1 Generalization-Specialization

2.2.5 Micro-Services Pattern

Provides for Scalability, modifiability, adaptability

Commented [MJK1]: Centrilization of logging, micro-services, service names, and mapping of all node information

Commented [MJK2]: Include statelessness as a goal for the architecture. Scalability

2.2.5.1 APIs, methods, and data tied to a domain or bounded-context

2.2.6 Shared Data Repository Pattern

Provides for scalability, modifiability, adaptability

2.2.6.1 Centralize customer profile and configuration data

2.2.6.2 Centralized event data

2.2.6.3 Data services APIs (no direct access to repositories)

2.3 System Concept Architecture

Commented [MJK3]: Batch processing may live on the edge for customers. What batch processing services can be broken across the Edge - Core bondaries?

Commented [MJK4]: Add security services into concept architecture.

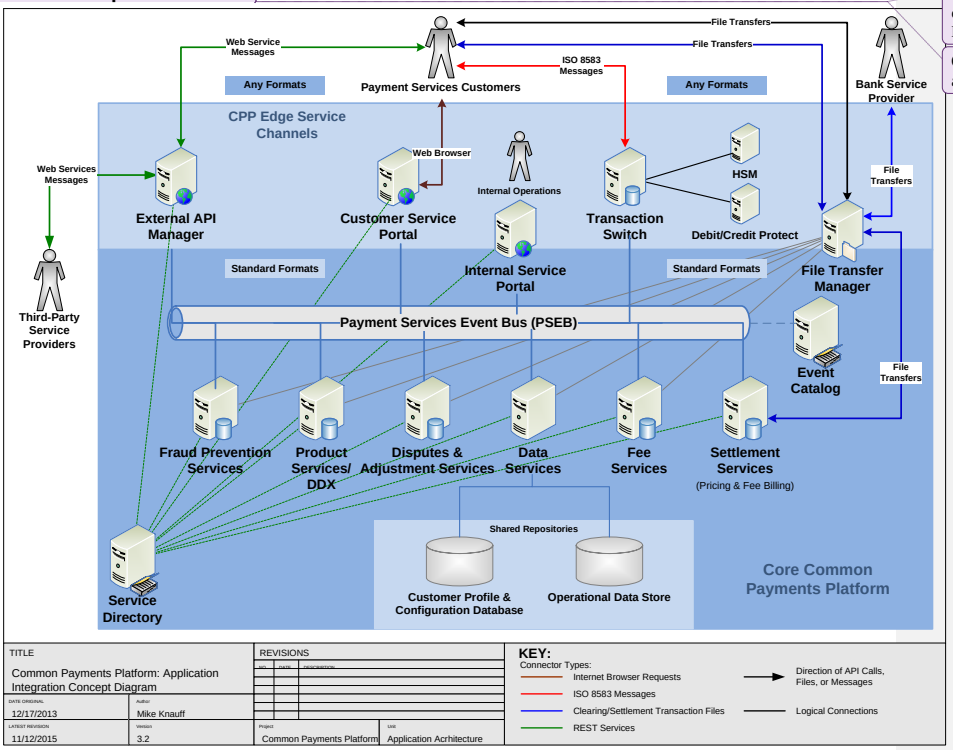


Figure 1 - CPP Platform Concept Diagram

2.3.1 Description

The *CPP* concept architecture makes use of the following logical components to provide support for real-time, near real-time, and offline integration of service consumers and service providers and to keep them loosely coupled to each other and the underlying technology infrastructure:

2.3.1.1 Service Directory

The *Service Directory* provides the ability for service consumers to dynamically discover available service providers without having to be coupled directly to the service provider's location or address (space coupling). The *Service Directory* holds the physical addresses of multiple instances of each type of service provider for REST service calls. Service providers utilize a canonical format for service calls, which allows service providers and service consumers to more easily integrate for multiple platform services.

Service providers register themselves with the Service Directory and establish heartbeat messages with the *Service Directory* so that the *Service Directory* can remove failed or unavailable service provider instances out of the directory. Service consumers utilize a logical name to find multiple available instances of a service provider registered in the *Service Directory*.

The *Service Directory* provides loose coupling between service providers and service consumers that require synchronous (real-time) service calls.

2.3.1.2 Payment Service Event Bus (PSEB)

The *Payment Services Event Bus (PSEB)* provides for the integration of publishers and subscribers for near real-time or asynchronous events. Publishers register their event types with an Event Catalog that provides a registry of event types (topics) and their definition. Clients that wish to subscribe to these events register themselves with the *PSEB* for receiving events related to specific topics. Events follow a canonical format so the publishers and subscribers can more easily integrate over a common communication channel.

The publish-subscribe pattern of the *PSEB* keeps publishers and subscribers loosely coupled to each other since publisher do not need to know the identity of subscribers and subscribers do not need to know the identity of publishers. The publish-subscribe model of integration keeps parties loosely coupled from an identity perspective (space coupling), and the asynchronous communication allows publishers and subscribers to be available at different times (time coupling).

2.3.1.3 File Transfer Manager

The *File Transfer Manager* brokers all offline communication between service providers and service consumers. The *File Transfer Manager* provides a file directory structure and file store-and-forward infrastructure that allows service consumers to send and/or receive data files from service providers, and service providers to receive data files for invoking services offline and in a batch mode and then provide a response back to the service consumer via the *File Transfer Manager*.

The file store-and-forward method of integration provides loose coupling between service providers and service consumers since they do not have to be available at the same time (time coupling). The identity of service providers and service consumers are moderately coupled since both only need to know the directory in which to send and/or receive files (space coupling).

2.3.1.4 External API Manager

The *External API Manager* provides for the opaqueness of service provider location for service consumers that are located externally to the CPP. The *External API Manager* brokers service calls between externally located service consumers and service providers that may be externally located (vendor-provided) to the CPP or provided the CPP-proper.

The *External API Manager* allows vendor-provided services to be implemented as part of the CPP platform in a way that is completely hidden from the service consumer. The *External API Manager* allows the loose coupling between the service provider and the service consumer for REST APIs, as well as centralized API management, security, and governance.

2.3.1.5 Data Services

The *Data Services* component provides a service layer over the underlying technology database technology and database-specific formats and structures. *Data Services* provides a canonical model and definition for all common data on the model and provides a centralized location for platform customers and events. All platform service components and consumers integrate over the canonical data definitions and formats through a discrete service API. *Data Services* provides a common service and repository for common customer profile and configuration data (Customer Profile and Configuration Database (CPCD)) and a common data sink for all platform events and processing data (Operational Data Store (ODS)).

The *Data Services* provides for the following:

- Easier integration of data producers and data consumers through the use of canonical forms of customer profile and event data
- Loose coupling between data producers/data consumers and the underlying data technology and data structures through the use of an abstracted service API

- Loose coupling between producers of the data and consumers of the data in both identity (space coupling) and availability (time coupling)

3. Architectural Quality Requirements

3.1 Availability

3.1.1 Requirements

Key infrastructure components that provide support services to the business services and applications must be highly-available, i.e. essentially always available. Other service availability can vary by business requirement but all services should be available during service/component maintenance. The following are the minimum availability requirements.

3.1.1.1 Critical Services/Components (99.999% Available)

- CPCD Database
- Data Services: Customer Profile Services
- Data Tokenization
- Data Encryption
- PaaS/Pivotal Cloud Foundry
- Payment Services Event Bus (PSEB)
- Security Services for user/system authentication, authorization, and access control
- Data Services: User Credentials
- Data Tokenization Proxy Application Service
- Layer 7 API Broker
- Service Directory

Commented [MJK5]: How will Layer 7 work with Angular JS front-ends with SSO? Will use a cookie instead of a JWT token to authenticate users? Don't want to authenticate for each service call.

3.1.1.2 Important Services/Components (99.9% Available)

- Concourse CPP Adapter
- Customer Portal
- Data Services: Events
- Data Services: ODS Near-Real Time (Horton Works Stack/Hadoop)
- Data Services: Transactions
- Dispute Application Services
- Dispute Database: Oracle Exadata
- Document/Image Service
- Data Vault (Documentum for now)
- File Transfer Manager: DBV
- Operations Portal

Commented [MJK6]: May need to add a data vault for data that cannot be tokenized and needs to be encrypted.

3.1.1.3 Other Services/Components (99.8% Available)

- Logging Service
- Service Monitor

3.1.1.4 Active-Active Node Configuration

The CPP services shall support redundant components in an active-mode whenever possible. Redundant components in a passive-mode shall support automated fail-over to the degree possible. Components in a redundant passive mode shall support the following recovery point and time objectives:

- Recovery Point Objectives (RPO) of 0 minutes
- Recovery Time Objectives (RTO) of 20 minutes or less

3.1.1.5 No Impact to Availability caused by Maintenance Activities

The CPP services shall remain available for service or service component upgrades. The CPP services shall be design to allow for incremental upgrades across service nodes so that all nodes do not need to be brought down to support upgrades to services or service infrastructure components.

3.2 Scalability and Performance

3.2.1 Scalability

The CPP Services shall scale both vertically (larger nodes with more resources) and horizontally (more nodes) so that performance can be maintained at a constant level. Horizontal scalability is required for PaaS deployments. Applications shall be designed so that processing resource requirements shall increase linearly with increasing volumes.

The CPP processes shall hold important performance and latency requirements constant as transaction volume increases by scaling the runtime implementation with additional processes and nodes

3.2.2 Performance

Commented [MJK7]: May need to tweak based upon real world learnings. Primary services may need to be aggressive.

3.2.2.1 Real-Time Primary Services

Commented [MJK8]: Change the wording initial guidelines and that will change based upon invidiual use-case requirements.

Real-time service platforms that provide direct access to resources shall be able to process a single transaction, request and response message, and excluding network latency in the following timeframes:

- Tier 1 (< 125ms) 80%
- Tier 2 (< 250ms) 95%
- Tier 3 (< 500ms) 97%
- Tier 4 (< 1000ms) 99.999%

3.2.2.2 Real-Time Aggregate Services

Real-time service platforms that aggregate multiple primary services and provide additional processing on top of these services shall be able to process a single transaction, request and response message, and excluding network latency in the following timeframes:

- Tier 1 (< 375ms) 80%
- Tier 2 (< 750ms) 95%
- Tier 3 (< 1500ms) 97%
- Tier 4 (< 3000ms) 99.999%

3.3 Modifiability

The CPP Services shall be adaptable to changing requirements in the external business environment and internal operating environment.

3.3.1 Extensibility

The CPP Services shall be able to accommodate new requirements in business logic and presentation of results with a minimum of changes to source code and underlying data structures.

The CPP Services shall not propagate changes to internal data structures or underlying technologies to other platform components.

The CPP Services shall be able to provide primary application services and components that are reusable for other applications and business functions.

The CPP Services shall provide integration points for vendor-provided services in a way that is seamless to customers between vendor-provided services and services that are native to the platform.

3.3.2 Adaptability

The CPP Services shall be able to accommodate changes to new operating environments (hardware and system software including PaaS environments) without having to change source code. Changes to middleware shall result in a minimum of modifications to source code, which are localized to a few or a single abstracting component through the use of standard API's.

The CPP Services shall be able to accommodate new service requestors or service providers with no changes to source code or minimum changes to the source code. The addition of new service requestors may result in the addition of new service nodes to handle the additional volume.

The CPP Services shall be able to add volume and service requestors with no degradation in system performance and response times.

3.4 Security

3.4.1 Requirements

Commented [MJK9]: Add encryption of sensitive data in-flight.

The CPP Services shall be PCI-DSS compliant to enable transactions and data that contain PCI-sensitive data owned by Diners Club, Discover, MasterCard, Visa, and other payment schemes to run securely over the same applications and application infrastructure. The CPP Services shall maintain availability SLAs for legitimate use in spite of denial of service attacks or other malicious attempts at disruption of platform services.

3.4.1.1 Data Tokenization

The CPP Services shall tokenize PCI-sensitive data at the earliest opportunity within the platform deployment layers in order to reduce the impact to the number of systems and components that fall under PCI-DSS hardening.

3.4.1.2 Prevention of Unauthorized Access

The CPP Services shall provide confidentiality of data and services through protection of unauthorized access.

- The services shall make use of requestor authentication, authorization, and access control to ensure that only authenticated and authorized users and services view or modify platform data
- The services shall provide integrity of the data and services that are being delivered from unauthorized modification during processing and delivery
- The services shall provide assurance and verification of all parties involved in a service request
- The services shall provide non-repudiation of changes made on the platform as well as service requests that occur over the platform
- The services shall track (audit) system and user activities at a level that can be used to reconstruct them for forensic purposes

3.5 Testability

The CPP Services shall provide the ability to test the various services and service components at different levels of granularity, including unit or sub-component, component, service, and application levels. Services and applications will provide interfaces that allow mock or test implementations to be substituted at run time.

- The CPP Services shall enable automated component level testing
- The CPP Services shall provide components with the ability for testers and/or test harnesses to control inputs and internal state, and to observe the components' outputs for correctness in cases where white box testing is either required or desired
- The CPP Services shall allow for automated test tools and automated regression testing
- Component level (individual units), system level (service level), and integration (grouped services)
  - Automated test execution, test results
- Automated QA tests and reports

Commented [MJK10]: Firther define and expand

3.6 Constraints

3.6.1 Schedule (Time-to-Market)

The CPP shall be deployed initially into production no later than 4th Quarter, 2016. Design and implementation decisions may be made that run contrary to the architecture requirements and design in order to realize the desired schedule. However this document should be viewed as the strategic direction of the platform.

3.6.2 Cost

The CPP architecture is constrained by the original CPP Phase I budget that was approved by the Discover Investment Council in 2014.

3.6.3 Product

The CPP architecture qualities, features, and functions will be traded as appropriate against schedule and cost pressures. These trades-off will be made and agreed to by the primary CPP stakeholders, which are as of the current date of this document:

- Scott DeBoard, Vice-President, Payment Services Operations (Project Owner)
- Tony Zeis, Vice-President, Payment Services Operations (Technology Project Owner)
- Steve Crider, Director, CPP Delivery
- Ewen McPherson, Director, CPP Software Engineering

4. Size and Performance

4.1 PSEB and Event Subscriber Scale Estimates

- 2000 TPS sustained
- 4000 TPS at peak

4.2 ODS Primary Data Size Estimates

- (30 MM trans/day[1] * 10 KB/trans) * 365 days/year) * 5 years ≈ 550 TB over 5 years

[1] Estimate includes Diners Club, Discover, and PULSE Network authorization and clearing/settlement transactions

5. Use-Case View

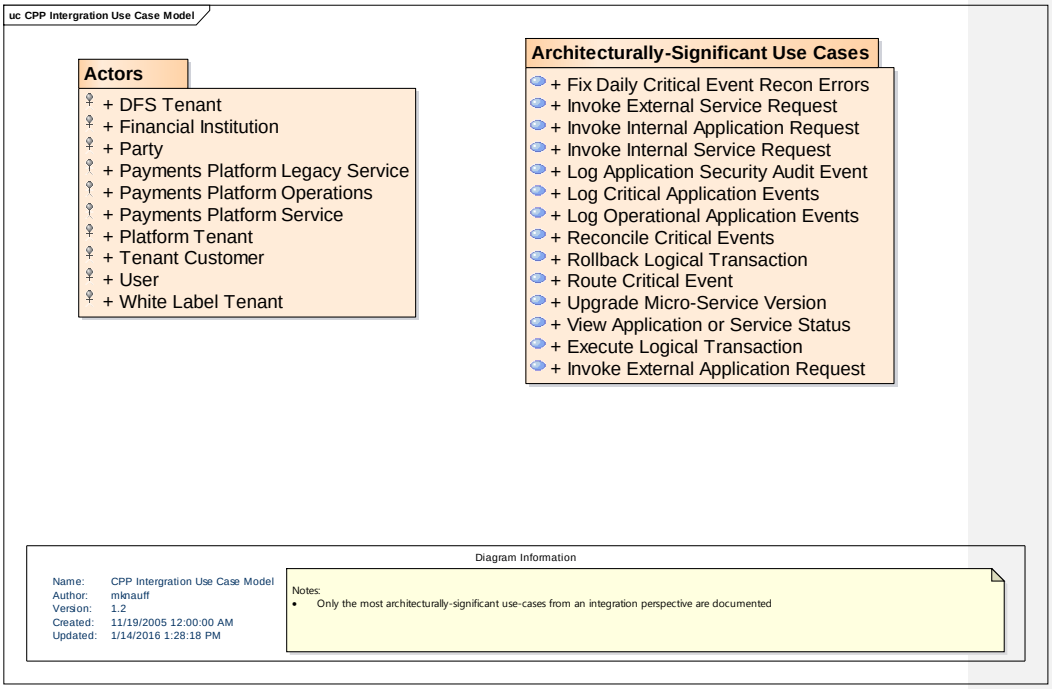


Figure 2 - Architecturally-Significant Use Cases

5.1 Use-Case Realizations

5.1.1 Execute Logical (Business) Transaction

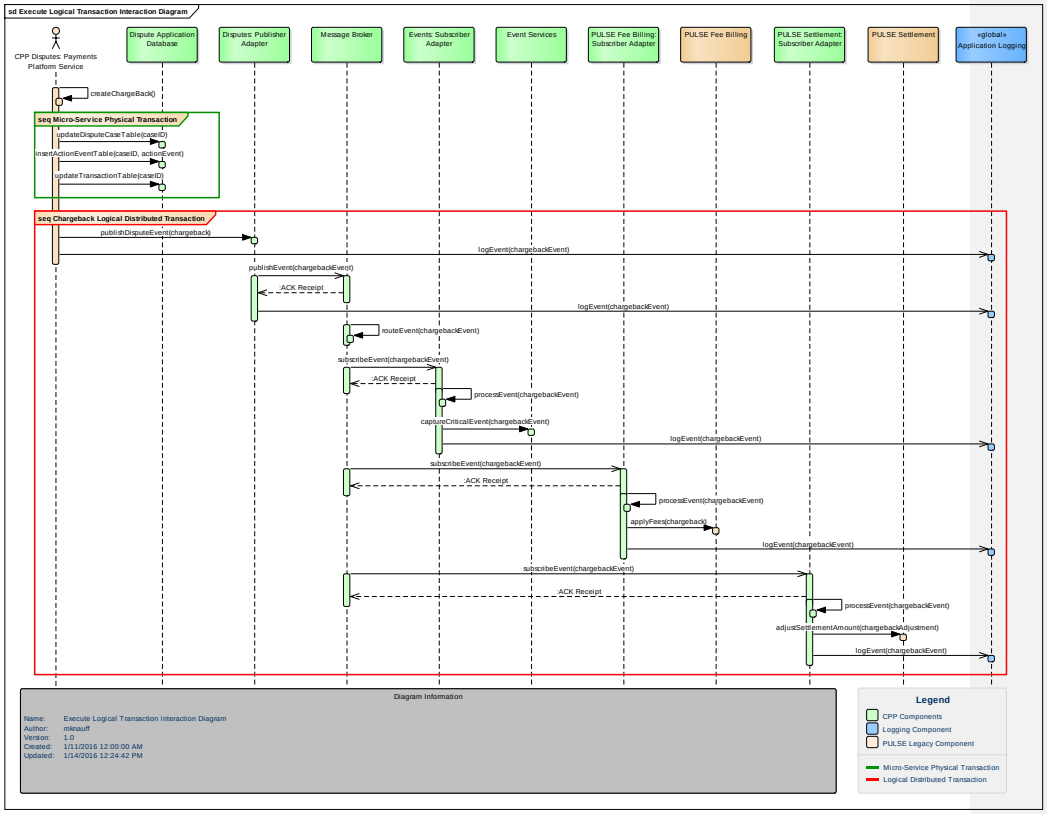
5.1.1.1 Brief Description

Execute Logical (Business) Transaction allows applications to call multiple services as part of a single, logical transaction and ensure that all platform state affected by the transaction are eventually made consistent.

5.1.1.2 Pre-Conditions

- All service providers are Idempotent
- All service providers support an asynchronous subscription messaging interface via the PSEB
- Service requests have a message type defined as Critical
- All messages that are part of the logical transaction have a unique correlation ID or transaction ID that can identify all messages that are part of the same logical transaction
- Subscribers are designated as either required or not-required for receipt of the event
    - Subscribers designated as required are considered part of the logical transaction during event reconciliation processing
- The Message Broker has been configured to route the service requests to the appropriate service providers that are part of the logical transaction

5.1.1.3 Interaction Diagram



5.1.1.4 Flow of Events

1. A chargeback dispute action gets created via the CPP Disputes Micro-Service.
2. The Disputes Micro-Service creates a physical transaction to update the Dispute Application Database, which is the domain database (bounded context) of the Dispute Micro-Service. The Dispute Micro-Service utilizes a physical transaction to enforce ACID properties when updating multiple tables for a dispute case in a multi-user and multi-process environment.
3. The Disputes Micro-Service create a logical distributed transaction for the chargeback send the chargeback request to the Disputes Publisher Adapter, which by publishes the chargeback event to the Message Broker. The Disputes Micro-Service logs the chargeback event for event reconciliation processing and integrity of the logical transaction.
4. The Dispute Publisher Adapter places the chargeback request into the canonical format for events and publishes the event to the Message Broker.
5. The Message Broker determines all subscribers of the event:
    - Event Services

- PULSE Fee Billing
- PULSE Settlement

6. The Message Broker routes the chargeback event to all registered subscribers.
7. The Events Subscriber Adapter (required) receives the chargeback event in the canonical format and forwards it to Event Services for processing. Receipt of the chargeback event is logged for event reconciliation processing and integrity of the logical transaction.
8. The Events Micro-Service captures the critical chargeback event for storage and logs the receipt and processing of the event for event reconciliation processing and integrity of the logical transaction.
9. The PULSE Fee Billing Subscriber Adapter (required) receives the chargeback event in the canonical format, transforms it into the PULSE Fee Billing proprietary format, and forwards it to the PULSE Fee Billing application for processing. Receipt of the chargeback event is logged for event reconciliation processing and integrity of the logical transaction.
10. The PULSE Fee Billing Application captures the chargeback event for fee calculations and fee settlement.
11. The PULSE Settlement Subscriber Adapter (required) receives the chargeback event in the canonical format, transforms it into the PULSE Settlement proprietary format, and forwards it to the PULSE Settlement application for processing. Receipt of the chargeback event is logged for event reconciliation processing and integrity of the logical transaction.
12. The PULSE Settlement Application captures the chargeback event for adjustment of the settlement amount and settlement of funds.

5.1.1.5 Post Conditions

- All PSEB components (publisher, subscribers, Message Broker) have all successfully acknowledged receipt of the event
- All CPP components involved in the logical transaction have successfully logged their event processing with the same event correlation or transaction ID (see Log Critical Application Events)
- The CPP Data Services has run event reconciliation to ensure logical transaction integrity (see Reconcile Critical Events)
- CPP Operations has performed Fix Daily Critical Event Recon Errors to ensure eventual consistency of the transaction data

Formatted: Underline, Font color: Blue

Formatted: Underline, Font color: Blue

5.1.1.6 Notes

1. Data that must be maintained as highly consistent at any point in time should be made part of the same data domain or context and have a single micro-service that controls access to the data and that utilizes transaction mechanisms to ensure ACID properties are maintained for the data.
2. Open Question: Do we consider that all operations that are part of a logical unit of work and that span across more than one (1) service domain (micro-service) be sent as asynchronous messages?
    a. What about the scaling issue within a single domain?
        i. All data or entities cannot fit within a single cluster
        ii. How is transactional integrity maintained?
            1. Sharding based upon Entities? Each Entity is the transactional scope and fits within a single machine or cluster
    b. Do we need cancelling and confirming message support?

i. Just the support of canceling if an operation is tentative or if an operation needs to be reversed?

ii. A confirming message is required only if an operation is inherently tentative to begin with?

c. Do we need a middleware construct that allows the application to call a "transaction" that automatically uses asynchronous messaging under the covers to implement the transation?

3. How do we provide "localization" of logical transaction participants for maintainability?

a. Current approach requires that logical transaction participants by configured as part of the event routing on the bus:

i. Identification of subscribers for routing

ii. Identification of which subscribers are required as part of logical transaction

iii. Each subscribers determines what to do with the event

iv. Identification of message as "critical" or "transactional"

b. Another approach is to send "command" messages to specific application queues t explicitly define the endpoints in the transaction and explicitly state what is to be done

i. Tightly couples participants but localizes the transaction logic within a specific component's source code

5.1.2 Fix Daily Critical Event Recon Errors

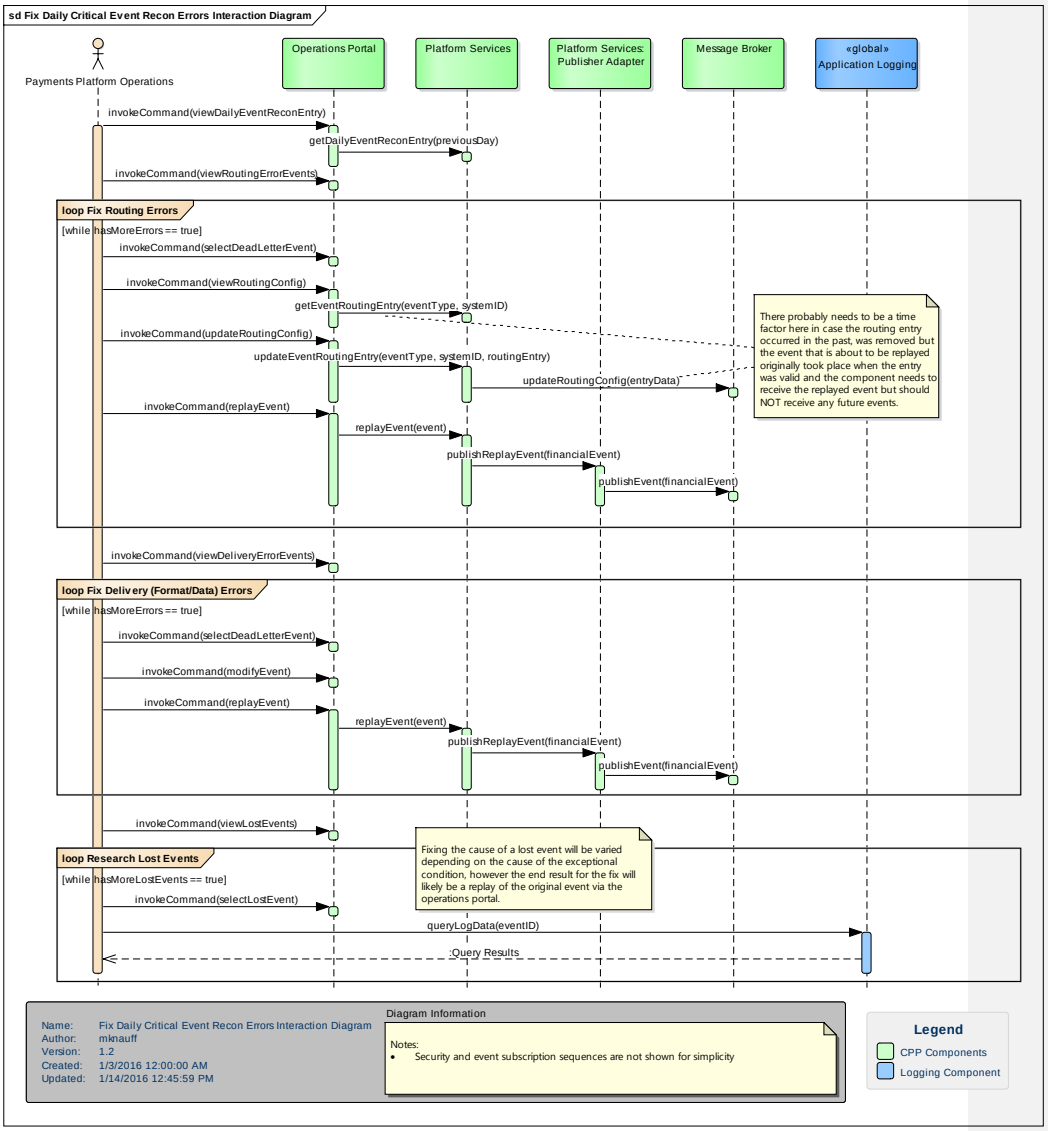
5.1.2.1 Brief Description

Fix Daily Critical Event Recon Error provides the Payments Platform Operations Group with the ability to determine what financial events failed to complete, research the cause, and resubmit the events for successful completion of the event processing.

5.1.2.2 Pre-Conditions

- The Daily Critical Event Recon Entry has been successfully been created with Platform Services
- All event subscribers are Idempotent
    - Subscribing components must be able to ignore duplicate events in the case of replayed events OR replayed events must only target subscribers that have not successfully received the event previously

5.1.2.3 Interaction Diagram



TODO: I had a republish flow under Fix Critical Event Reconciliation Errors. I misunderstood and didn't realize it was for a failed ACKs from the PSEB. Seems like republish event flow could be an adapter library function but the library would need to hold the event in some type of durable storage and would need to figure out how many retries before raising some type of exception and placing it into its log. Once in the log it could be a platform service that looks for failed publish events and allows a function from the Operations Portal to republish or replay the event from an ODS process that gathers publish errors from the centralized logging service as part of the event reconciliation process (find failed publishing events) in addition to successfully published events that did not successfully make it to all subscribers and/or were not successfully processed by the subscribers.

5.1.2.4 Flow of Events

1. The *Payment Platform Operations* user requests that the Operations Portal display the Daily Event Recon Entry for financial events.
2. The Operations Portal sends a request to Platform Services to get the Daily Critical Event Recon Entry for the previous day.
3. The *Payment Platform Operations* user requests the Operations Portal to display the Dead Letter Events – Routing Errors from the Daily Event Recon Entry.
4. The *Payment Platform Operations* user selects a Dead Letter Event and performs the following for each Dead Letter – Routing Error event:
    a) Retrieves the routing entry from Platform Services for the event
    b) Identifies the error the in the routing configuration and updates the routing configuration entry
    c) Submits the updated routing configuration information to Platform Services
    d) Platform Services updates the master entry and then updates the Message Broker's routing configuration
    e) Replays the event so that the component(s) that did not receive the event receive from the Message Broker, which has the updated and corrected routing information
5. The *Payment Platform Operations* user requests the Operations Portal to display the Dead Letter Events – Delivery Error Events from the Daily Event Recon Entry.
6. The *Payment Platform Operations* user selects a Dead Letter Event – Delivery Error and performs the following for each Error event:
    a) Determines the formatting or data error for the event
    b) Modifies the event to correct the error
    c) Submits the updated event data to be replayed by Platform Services
    d) Platform Services submits the updated event data for publishing to its publishing adapter, which forwards the publishing request to the Message Broker
    e) The Message Broker publishes the event so that the component(s) could not process the event do so with the updated format and/or data
7. The *Payment Platform Operations* user requests the Operations Portal to display the Lost Events from the Daily Event Recon Entry.
8. The *Payment Platform Operations* user selects a Lost Event and performs the following for each Lost event:
    a) Opens up the Central Application Logging Tool and submits a query to return all application log data that contains the unique event ID of the lost event

b) Utilizes the log data to determine the cause of the exceptional condition(s) that caused the event not to complete all routing processing

c) Contacts the appropriate groups to correct the error(s)

d) Replays the event at the appropriate time after all exceptional conditions have been fixed

5.1.2.5 Post Conditions

- All Dead Letter Event issues have been fixed and the corresponding financial events have been replayed
- All Lost Events have been researched through use of the centralized log data, exceptional conditions have been identified, and the corresponding stakeholders have been contacted
- Previously identified Lost Events that have been fixed have been successfully replayed

5.1.2.6 Notes

1. Lost events will require further analysis in order to identify detailed causes of Lost Events and the tools and processes that will be required to fix these errors.
2. Additional information needs to be gathered on how fixes to Lost Events will be applied and then communicated back to Operations in order to reply the event.
3. Sensitivity of the critical/financial event may dictate a manual update to the affected system such as settlement or fee billing.

5.1.3 Log Critical Application Events

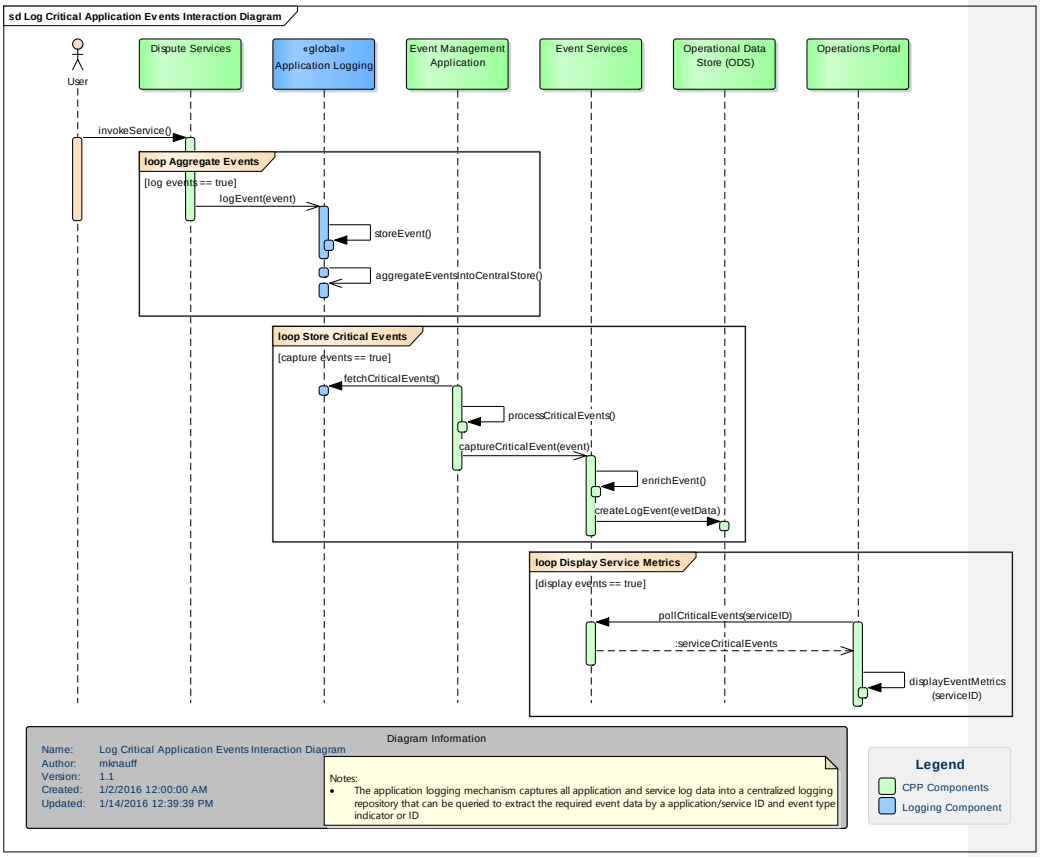
5.1.3.1 Brief Description

Log Critical Application Events provides applications and services with the ability to enter critical application events into a log file with tags that allows this information to be collected by a centralized logging system and placed into data sink. Once in the data sink these events can be utilized by important platform processes like event reconciliation or monitoring.

5.1.3.2 Pre-Conditions

- The application or service has logging enabled
- The application or service logging data conforms to the canonical logging format and log event classifications
- Each application or service has a unique application or service ID that can be used to uniquely identify its log data when the log data is aggregated into a centralized log store

5.1.3.3 Interaction Diagram



5.1.3.4 Flow of Events

1. An application or service receives a request to execute or invoke a service.
2. The Service creates log entry/events that conform to the canonical form for all CPP log entries/events, which includes the unique application/service identifier, the log event type, and the canonical elements for the log event type.
3. The Service requests that the log event be captured by the Application Logging Service.
4. The Application Logging Service captures and stores the event.
5. The Application Logging Service aggregates all application/service events into a centralized logging repository.
6. The ODS Batch and Reporting Application periodically collects all critical CPP application/service events from the centralized log repository by making fetch requests to the logging service.
7. The ODS Batch and Reporting Application processes the critical events and performs any required filtering, data enrichment, transformations, summarization, etc. and sends this data to Event Service.
8. The Event Service provides any further event enrichment and stores the log event data in the ODS.
9. The Operations Portal Application polls Event Services for any new critical events that have occurred across the Common Payments Platform.
10. The Operations Portal Application sends request to Event Services to retrieve critical events based upon unique application or service IDs.
11. The Operations Portal Application receives the critical event data and metrics and displays these in Operations Portal Dashboard.

5.1.3.5 Post Conditions

- All application/service events are tagged appropriately and stored in the central logging repository
- All critical application/service events have been processed and properly enriched and summarized when appropriate within the ODS
- All critical application/service events and metrics subscribed to by operations' users have been displayed within the appropriate Operations Portal Dashboards

5.1.3.6 Notes

1. Critical application events are considered to fit into one or more of the following categories:
    - Key Performance Indicators (KPIs)
        - Availability
        - Response Times (latency)
        - Load (volume)
    - Financial
        - Events the move funds or cause adjustments to funds settlement
        - Events that result in the assessment of fees
    - Transactional
        - Events that are part of a logical unit of work and that must all be successfully processed in order to maintain the platform in a consistent and correct state

5.1.4 Invoke External Application Request

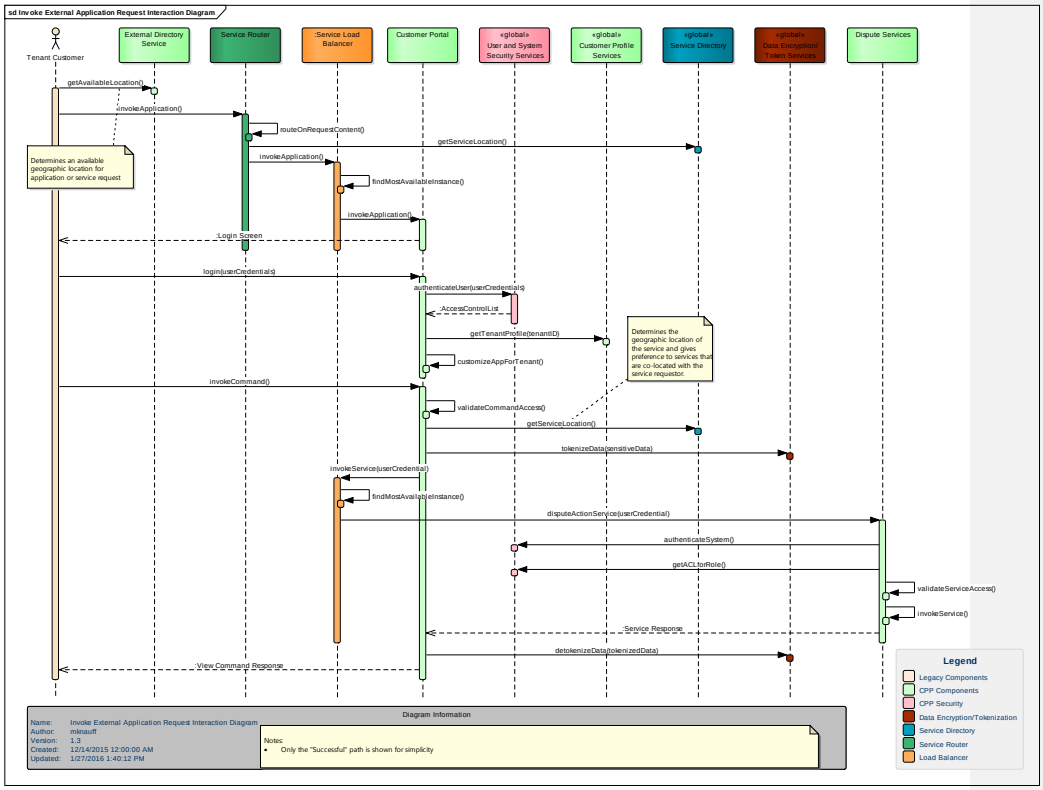
5.1.4.1 Brief Description

Invoke External Application Request allows *Tenant Customers* to gain access to platform applications and services.

5.1.4.2 Pre-Conditions

- The Tenant Customer has an authorized user credential or set of user credentials
- The Tenant Customer has been defined with a set of application access permissions
- Application and application service component instances are dynamically registered with the relevant application and service directories and that their state (availability and capacity) is monitored

5.1.4.3 Interaction Diagram



5.1.4.4 Flow of Events

1. The Tenant Customer (User) requests access to the Customer Portal (application) in the form of a Uniform Resource Identifier (URI).
2. The External Service Directory determines an available geographic location of the resource (application).

3. The External Service Directory returns an available location for the application.
4. The User sends a request to invoke the application.
5. A Router accepts the request, examines the message request and content and determines if it needs to apply any message content-based routing rules.
6. The Router calls the service directory to lookup the services based upon the content-based routing rules and then invokes the appropriate service.
7. A load balancer intercepts the request, determines the most available application instance, and invokes the application instance.
8. The User enters her/his credentials into the application.
9. The Application authenticates the User and returns the Tenant/Tenant Customer that the User belongs to and the access control list for the user and/or the user's role.
10. The Application fetches the tenant profile from the Customer Profile Service.
11. The Application customizes the branding and look and feel based upon the Tenant/Tenant Customer profile that the User belongs to.
12. The User invokes an Application command or service.
13. The Application validates that the User has the appropriate permission to invoke the service.
14. The Application sends a request to the Service Directory for the location of the service.
15. The Service Directory provides an available location for the service giving preference to the same location as the Application instance.
16. The Customer Portal sends a request to the Data Encryption/Token Service to tokenize the sensitive data before invoking the Micro-Service for Disputes.
17. The Customer Portal invokes the Dispute Service with tokenized data and utilizing the location provided by the Directory Service.
18. A load balancer intercepts the request, determines the most available service instance, and invokes the service instance (dispute service as an example).
19. The Application Service validates the identity of the system making the service request and that the system and user role have access rights to the service.
20. The Application Service Component performs the service and returns the results to the Customer Portal.
21. The Customer Portal invokes the Data Encryption/Token Service to De-tokenize the sensitive data in order to display to the Tenant Customer.

5.1.4.5 Post Conditions

- The application service request is successfully executed by the application and dependent application service components

5.1.4.6 Notes

1. Applications and application service components are geographically dispersed.
2. Application and application service directories or associated components maintain the following about each application and application service component:
    - The physical location of each instance
    - The state of each instance (healthy or un-healthy)
    - The current capacity or load of each instance

3. Application and service requests can be routed across geographic locations when necessary to achieve availability service levels but have a preference for "location affinity" under nominal conditions.
4. Application and service requests can be routed based upon the content of message headers and payloads.

5.1.5 Invoke External Service Request

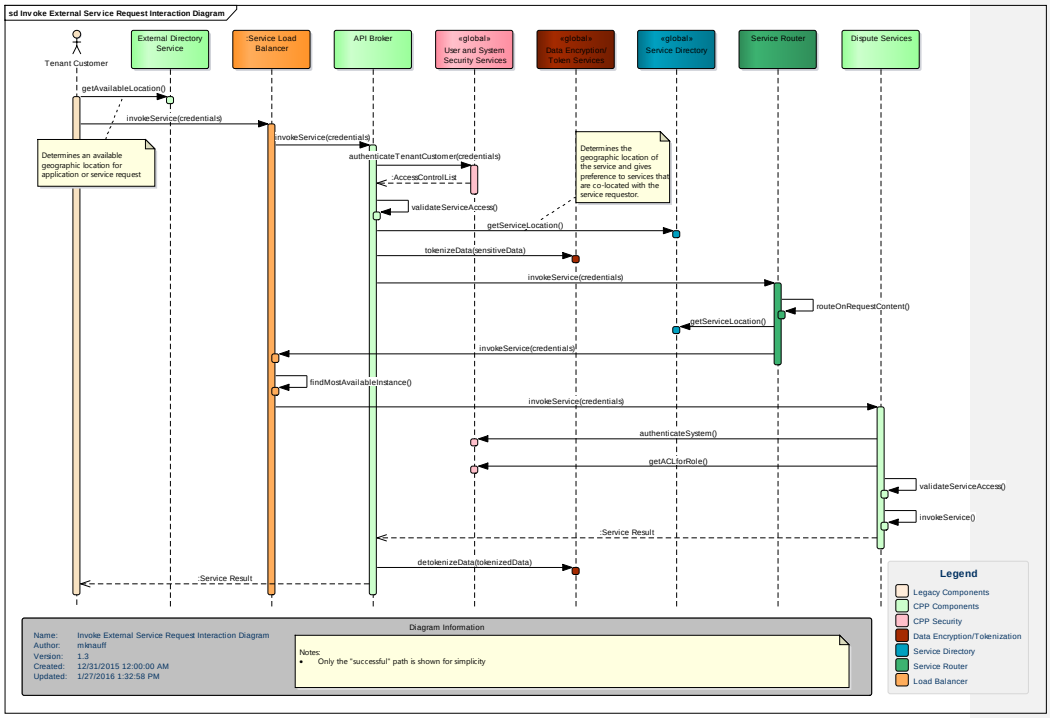
5.1.5.1 Brief Description

Invoke External Service Request allows *Tenant Customers* to call CPP services from their own applications.

5.1.5.2 Pre-Conditions

- The Tenant Customer has an authorized credential to access the requested service via the API Broker
- The API Broker and application service component instances are dynamically registered with the relevant service directories and that their state (availability and capacity) is monitored

5.1.5.3 Interaction Diagram



5.1.5.4 Flow of Events

1. The Tenant Customer (User) requests access to the service in the form of a Uniform Resource Identifier (URI).
2. The External Service Directory determines an available geographic location of the resource (service).
3. The External Service Directory returns an available location for the service.
4. The User sends a request to invoke the service with the tenant customer (user) credentials.
5. A load balancer intercepts the request, determines the most available API Broker instance, and invokes the broker instance.

6. The API Broker calls User and System Security Service to authenticate the user (Tenant Customer) and return the access control list for the user.
7. The API Broker validates that the user has the appropriate permission to invoke the service.
8. The API Broker sends a request to the Service Directory for the location of the service.
9. The Service Directory provides an available location for service.
10. The API Broker sends a request to the Data Encryption/Token Service to tokenize the sensitive data before invoking the Micro-Service for Disputes.
11. The API Broker invokes the Dispute Service with tokenized data and utilizing the location provided by the Directory Service.
12. A Router accepts the request, examines the message request and content and determines if it needs to apply any message content-based routing rules.
13. The Router calls the service directory to lookup the services based upon the content-based routing rules and then invokes the appropriate service.
14. A load balancer intercepts the request, determines the most available service instance, and invokes the service instance (dispute service as an example).
15. The Application Service validates the identity of the system making the service request and that the user has access rights to the service.
16. The Application Service Component performs the service and returns the result to the API Broker.
17. The API Broker invokes the Data Encryption/Token Service to De-tokenize the sensitive data in order to return the service result to the Tenant Customer.

5.1.5.5 Post Conditions

- The service request is successfully executed by the API Broker and dependent application service components

5.1.5.6 Notes

1. Applications and application service components are geographically dispersed.
2. Application and application service directories or associated components maintain the following about each application and application service component:
    - The physical location of each instance
    - The state of each instance (healthy or un-healthy)
    - The current capacity or load of each instance
3. Application and service requests can be routed across geographic locations when necessary to achieve availability service levels but have a preference for “location affinity” under nominal conditions.
4. Application and service requests can be routed based upon the content of message headers and payloads.

5.1.6 Invoke Internal Application Request

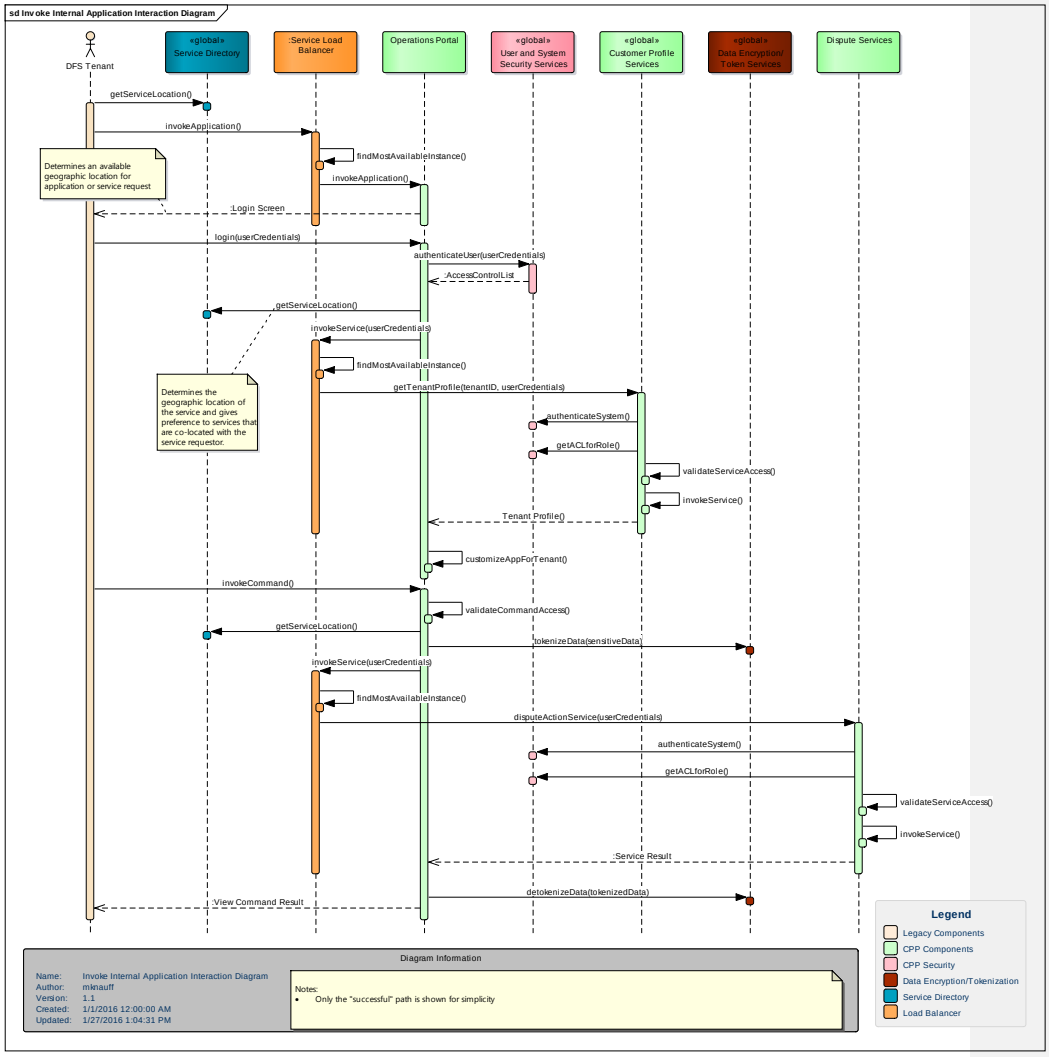
5.1.6.1 Brief Description

Invoke Internal Application Request allows *DFS (internal) Tenants* to gain access to platform applications and services.

5.1.6.2 Pre-Conditions

- The DFS Tenant has an authorized user credential or set of user credentials
- The DFS Tenant has been defined with a set of application access permissions
- Application and application service component instances are dynamically registered with the relevant application and service directories and that their state (availability and capacity) is monitored

5.1.6.3 Interaction Diagram



5.1.6.4 Flow of Events

1. The DFS Tenant (User) requests access to the Operations Portal (application) in the form of a Uniform Resource Identifier (URI).
2. The Service Directory determines an available geographic location of the resource (application).
3. The Service Directory returns an available location for the application.

4. The User sends a request to invoke the application.
5. A load balancer intercepts the request, determines the most available application instance, and invokes the application instance.
6. The User enters her/his credentials into the application.
7. The Application authenticates the User and returns the DFS Tenant that the User belongs to and the access control list for the user and/or the user's role.
8. The Application fetches the tenant profile from the Customer Profile Service.
9. The Application customizes the branding and look and feel based upon the DFS Tenant that the User belongs to.
10. The User invokes an Application command or service.
11. The Application validates that the User has the appropriate permission to invoke the service.
12. The Application sends a request to the Service Directory for the location of the service.
13. The Service Directory provides an available location for the service giving preference to the same location as the Operations Portal instance.
14. The Operations Portal sends a request to the Data Encryption/Token Service to tokenize the sensitive data before invoking the Micro-Service for Disputes.
15. A load balancer intercepts the request, determines the most available service instance, and invokes the service instance (dispute service as an example).
16. The Application Service validates the identity of the system making the service request and that the system and user role have access rights to the service.
17. The Application Service Component performs the service and returns the result to the Operations Portal.
18. The Operations Portal invokes the Data Encryption/Token Service to De-tokenize the sensitive data in order to display to the Tenant Customer.

5.1.6.5 Post Conditions

- The application service request is successfully executed by the application and dependent application service components

5.1.6.6 Notes

1. Applications and application service components are geographically dispersed.
2. Application and application service directories or associated components maintain the following about each application and application service component:
    - The physical location of each instance
    - The state of each instance (healthy or un-healthy)
    - The current capacity or load of each instance
3. Application and service requests can be routed across geographic locations when necessary to achieve availability service levels but have a preference for "location affinity" under nominal conditions.

5.1.7 Invoke Internal Service Request

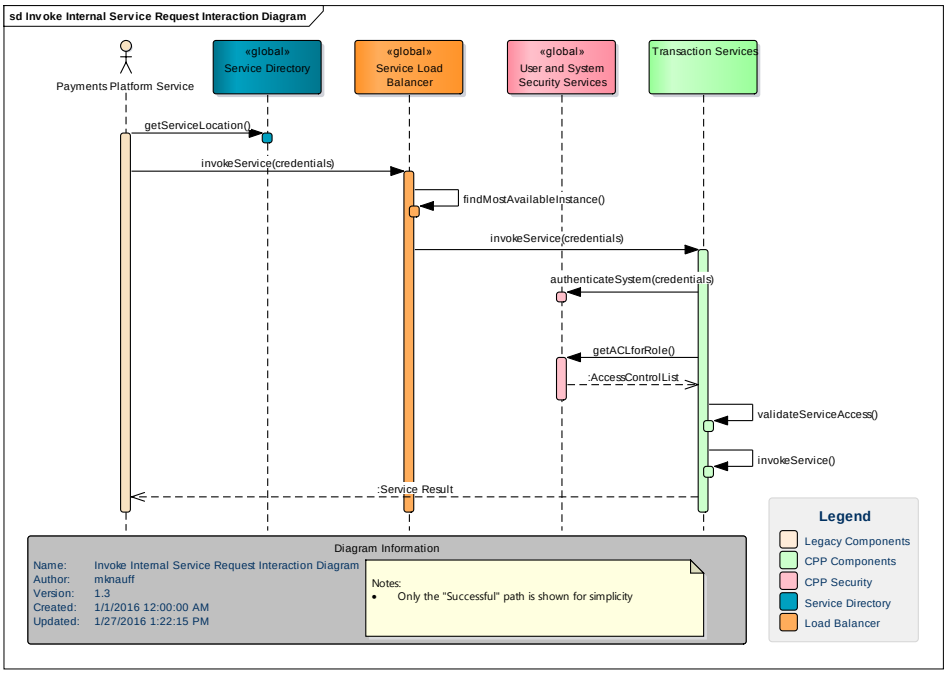
5.1.7.1 Brief Description

Invoke Internal Service Request allows platform applications and services to invoke other platform services.

5.1.7.2 Pre-Conditions

- The platform application or service has an authorized credential to access the requested service via the API Broker
- The requested service component instances are dynamically registered with the relevant service directories and that their state (availability and capacity) is monitored

5.1.7.3 Interaction Diagram



5.1.7.4 Flow of Events

1. The Payments Platform Service (User) requests access to the service in the form of a Uniform Resource Identifier (URI).
2. The Service Directory determines an available geographic location of the resource (service).
3. The Service Directory returns an available location for the service.
4. The User sends a request to invoke the service with the user and system credentials.
5. A load balancer intercepts the request, determines the most available service instance, and invokes the service instance.
6. The Service authenticates the user (Platform Service) and returns the access control list for the user.
7. The Service validates that the user has the appropriate permission to invoke the service.
8. The Service performs the requested service and returns the result.

5.1.7.5 Post Conditions

- The service request is successfully executed by the Service Provider and dependent application service components

5.1.7.6 Notes

1. Applications and application service components are geographically dispersed.
2. Application and application service directories or associated components maintain the following about each application and application service component:
   - The physical location of each instance
   - The state of each instance (healthy or un-healthy)
   - The current capacity or load of each instance
3. Application and service requests can be routed across geographic locations when necessary to achieve availability service levels but have a preference for “location affinity” under nominal conditions.
4. Assume system authentication and authorization occurs within the service similar to user authentication and that the service manages this.
   - What about system authentication provided by certificates?
5. Assume user authentication has occurred elsewhere but authorization/access control occurs within the service.

5.1.8 Reconcile Critical Events

5.1.8.1 Brief Description

Reconcile Critical Events ensures that all critical/financial events have been successfully transmitted to components that have a requirement to process the critical events and to determine if one of the following has occurred that prevented the event from being processed:

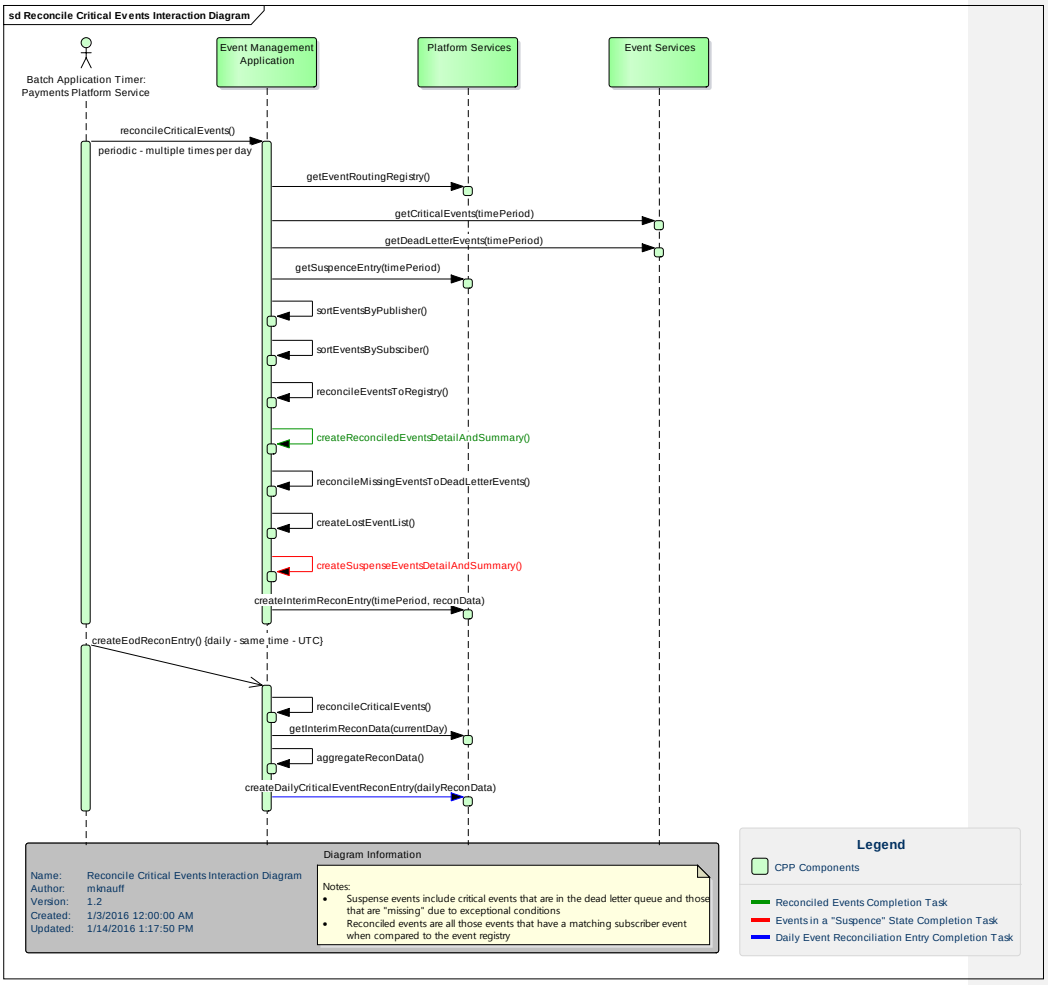
- The Critical Event could not be delivered by the Message Broker
  - The event resides within the Dead Letter Exchange → Routing or address error type
- The Critical Event could not be processed by the Subscriber
  - The event resides within the Dead Letter Exchange → Message format or data error type
- The Critical Event was lost during transmission somewhere between the publishing component and the subscribing component
  - The event resides within the log of the last component to successfully receive the event → Exceptional condition

Reconcile Critical Events balances the publishing and subscribing of all critical events, creates a suspense log of all outstanding critical events that reside within the Dead Letter Exchange, and creates alerts for "lost" critical events that need to be manually researched and corrected.

5.1.8.2 Pre-Conditions

- All platform components utilize a common "financial day" cut-off time in order to perform critical event reconciliation
- All platform components are synchronized to a common platform clock utilizing Universal Coordinated Time (UTC)
- All events contain a unique service/component ID and a unique event ID for the component that is handling the event as well as unique and common event ID (correlation ID) for all components that are processing the same event when it is written to each service/component's log
- A centralized configuration registry exists that maps all publishers to all subscribers for each critical event, and this registry is utilized to configure the Message Broker for the routing of events and the component for the reconciliation of these events

5.1.8.3 Interaction Diagram



5.1.8.4 Flow of Events

1. The Batch Application Timer Service sends a request periodically to reconcile critical events since the last reconciliation request.
2. The Event Management Application calls Platform Services to get the event routing registry.
3. The Event Management Application calls Event Services to get all of the Critical Events since the previous request.
4. The Event Management Application calls Event Services to get all of the Dead Letter Events since the last request.

5. The Event Management Application calls Platform Services to get the previous' interim recon suspense entry.
6. The Event Management Application sorts all of the critical events by publisher and also by subscriber and performs the following:
    - Checks that all published critical events have at least one (1) subscriber per event
    - Checks that all registered subscribers for the list of published critical events have a matching critical event
7. The Event Management Application checks previous day's suspense entry for any events with missing publishers.
8. The Event Management Application checks the Dead Letter Events for any missing subscriber events.
9. The Event Management Application creates a list of Lost Critical Events for any critical events that fail to appear in the list of reconciled events (critical events with matching subscribers) or in the list for dead letter events.
10. The Event Management Application creates an Interim Critical Recon Entry that includes the following information:
    - The list of reconciled critical events details and summary information
    - The list of "suspense" critical events details and summary information broken out into the following sub-categories:
        - Critical Events in a "Dead Letter" state
            - Routing Errors
            - Delivery Errors
        - Critical Events in a "Lost" state
11. The Batch Application Timer Service sends a request to reconcile critical events at the each day at the daily platform financial reconciliation cut-off time as designated in UTC to the Event Management Application
12. The Event Management Application first reconciles critical events since the last reconciliation request.
13. The Event Management Application sends a request to Platform Services to get all of the event recon entries for the current event recon day.
14. The Event Management Application aggregates all of recon entries for the day into a single recon entry that represents the critical reconciliation for the entire financial "day".
15. The Event Management Application sends a request to Platform Services to create the Daily Critical Event Recon Entry.

5.1.8.5 Post Conditions

- A Daily Critical Event Recon Entry has been created that clearly categorizes reconciled events and events in suspense for the following reasons; "Routing Error", "Delivery Error", and "Lost" and this information has been stored by Platform Services
- Critical Event Recon Detail and Summary Data can be retrieved by recon date, category/sub-category (Reconciled, Suspense, Suspense – Routing Error, Suspense – Delivery Error (Format and/or Data Error), Suspense – Lost (Exceptional Condition occurred), event type, component/service ID, etc.

5.1.8.6 Notes

1. Recon process involves asynchronous processing of critical events over the Payment Service Event Bus

- Recon Scope includes:
    - Application-specific PSEB Publishing Adapter
    - Message Broker
    - Application-specific PSEB Subscription Adapter

2. Individual source or sink components for critical events must perform their own reconciliation processing and error handling for internal critical event or transaction processing

5.1.9 Rollback/Cancel Logical (Business) Transaction

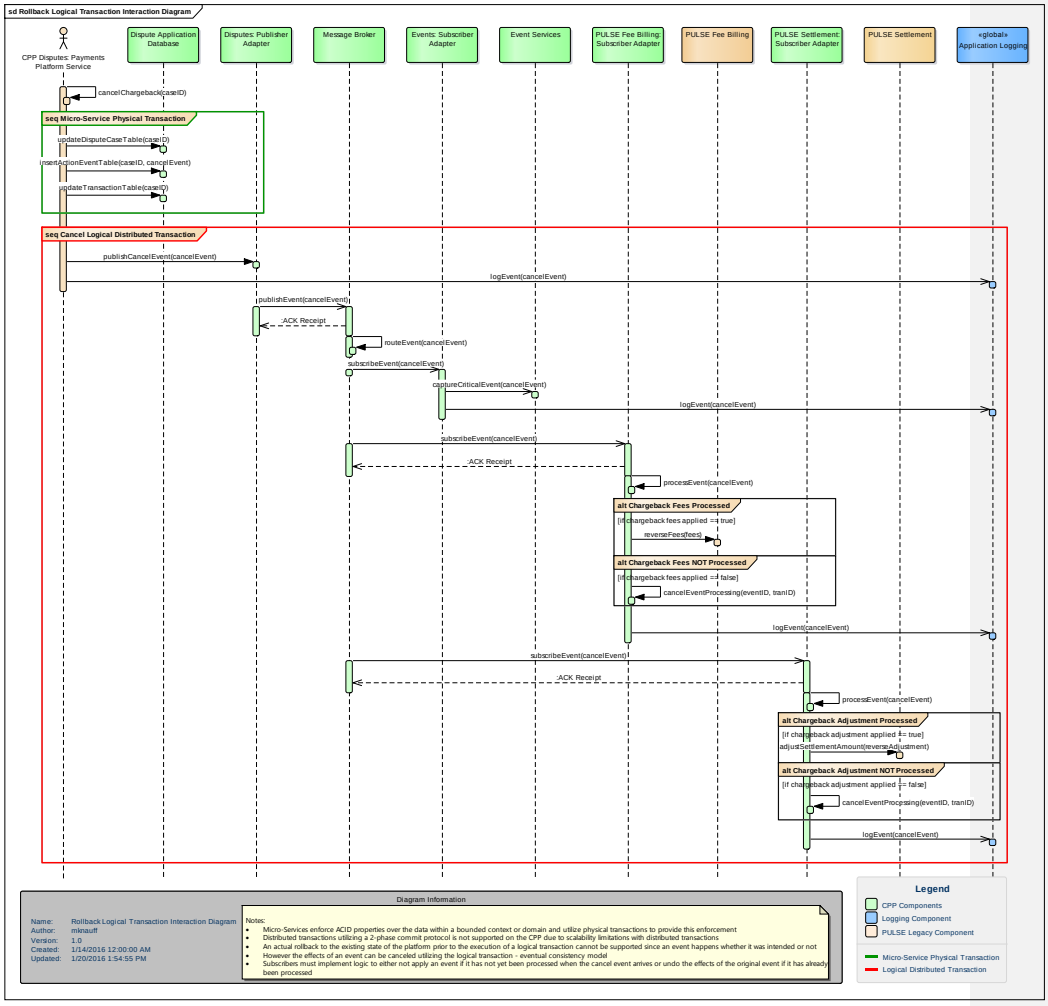
5.1.9.1 Brief Description

Rollback/Cancel Logical (Business) Transaction allows applications to cancel or "un-do" the effects of a previous logical transaction that affected the state of one or more services. The rollback does not erase or delete the previous events and so is not a true *rollback* in the traditional sense, but effectively cancels the previous event with the new *cancel* event. The rollback is itself a single logical transaction that ensures that the platform state is eventually made consistent as defined by the *cancel* event.

5.1.9.2 Pre-Conditions

- Service requests have a message type defined as Critical and a sub-type of *Cancel* or *Rollback*
- The *Cancel* message contains a new correlation ID or transaction ID for the logical rollback transaction and also the correlation ID or transaction ID of the original logical transaction
- The *Cancel* message contains the original message data (payload)
- Subscribers are designated as either required or not-required for receipt of the event
    - Subscribers designated as required are considered part of the logical transaction during event reconciliation processing
- The Message Broker has been configured to route the service requests to the appropriate service providers that are part of the logical transaction
- Subscribers are configured to handle *Cancel* messages and have implemented their own proprietary logic to reset their service or their customer state by removing the effects of the original event

5.1.9.3 Interaction Diagram



5.1.9.4 Flow of Events

1. A cancel dispute chargeback action gets created via the CPP Disputes Micro-Service.
2. The Disputes Micro-Service creates a physical transaction to update the Dispute Application Database so that the effects of the original chargeback action are reversed within the local domain database (bounded context) of the Dispute Micro-Service. The Dispute Micro-Service utilizes a physical transaction to enforce ACID properties when updating multiple tables for a dispute case in a multi-user and multi-process environment.

3. The Disputes Micro-Service creates a logical distributed transaction for the chargeback cancelation and sends the chargeback cancelation request to the Disputes Publisher Adapter, which publishes the chargeback cancelation event to the Message Broker. The Disputes Micro-Service logs the chargeback cancelation event for event reconciliation processing and integrity of the logical transaction.
4. The Dispute Publisher Adapter places the chargeback cancelation request into the canonical format for events and publishes the event to the Message Broker.
5. The Message Broker determines all subscribers of the original event:
    - Event Services
    - PULSE Fee Billing
    - PULSE Settlement
6. The Message Broker routes the chargeback cancelation event to all registered subscribers.
7. The Events Subscriber Adapter (required) receives the chargeback cancelation event in the canonical format and forwards it to Event Services for processing. Receipt of the chargeback cancelation event is logged for event reconciliation processing and integrity of the logical transaction.
8. The Events Micro-Service captures the critical chargeback cancelation event for storage and logs the receipt and processing of the event for event reconciliation processing and integrity of the logical transaction.
9. The PULSE Fee Billing Subscriber Adapter (required) receives the chargeback cancelation event in the canonical format and determines the following:

    If the original chargeback event was processed and the fees were applied then

    Send a request with the required data to the PULSE Fee Billing System to reverse the original fees.

    Else the original chargeback event was not processed by the PULSE Fee Billing System

    Cancel the original chargeback event processing and do not send the chargeback event to the PULSE Fee Billing System to have fees applied.

    The receipt of the chargeback event is logged for event reconciliation processing and integrity of the logical transaction.
10. The PULSE Settlement Subscriber Adapter (required) receives the chargeback cancelation event in the canonical format and determines the following:

    If the original chargeback event was processed and settlement adjustments were applied then

    Send a request with the required data to the PULSE Settlement System to reverse the original chargeback adjustments.

    Else the original chargeback event was not processed by the PULSE Settlement System

    Cancel the original chargeback event processing and do not send the chargeback event to the PULSE Settlement System to have settlement adjutsments applied.

    The receipt of the chargeback event is logged for event reconciliation processing and integrity of the logical transaction.

5.1.9.5 Post Conditions

- All PSEB components (publisher, subscribers, Message Broker) have all successfully acknowledged receipt of the cancel event

- All CPP components involved in the logical transaction have successfully logged their event processing with the same event correlation or transaction ID (see Log Critical Application Events)
- All CPP components involved in the logical transaction have successfully "unwound" the application and business effects of the original transaction event
- The CPP Data Services has run event reconciliation to ensure logical transaction integrity (see Reconcile Critical Events)
- CPP Operations has performed Fix Daily Critical Event Recon Errors to ensure eventual consistency of the transaction data

Formatted: Underline, Font color: Blue

Formatted: Underline, Font color: Blue

5.1.9.6 Notes

1. Data that must be maintained as highly consistent at any point in time should be made part of the same data domain or context and have a single micro-service that controls access to the data and that utilizes transaction mechanisms to ensure ACID properties are maintained for the data.
2. Components that support logical transactions must also implement the required logic to receive and process *cancel <event type>* transactions. This includes the required logic to know if an event has been processed and applied (service or application state change) or if it is still waiting to be processed.

   If an event has changed the service or application state then the component must be apply the appropriate business logic to reverse the business effects of the original event. If the original event is still waiting to be processed then the component must remove it from the processing queue[2] so that no service or application state changes occur.

[2] Reconciliation processing must take this into consideration; the reconciliation process may never see a logical transaction fully complete due to it being canceled by a subsequent cancel request.

5.1.10 Route Critical Event

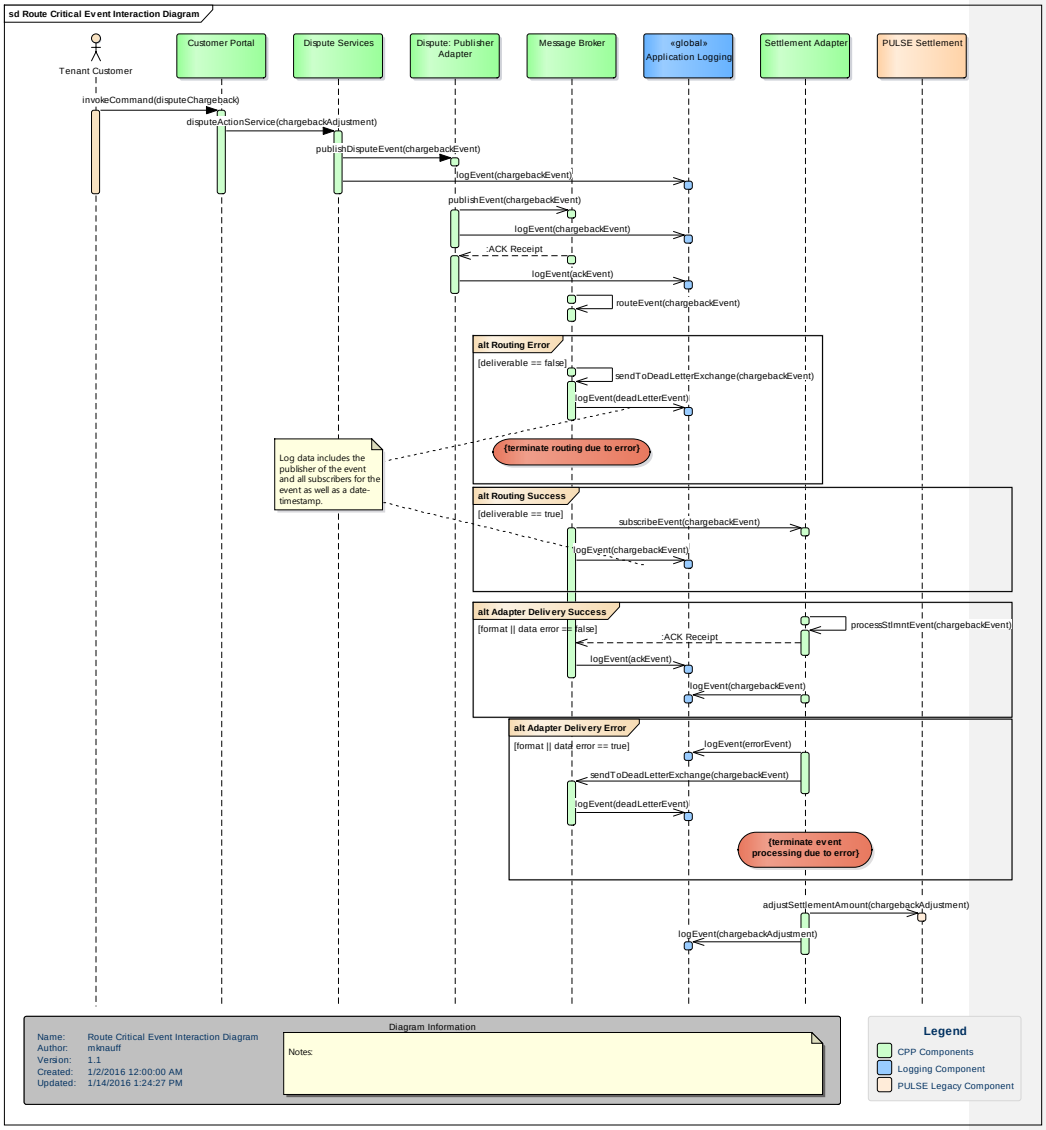
5.1.10.1 Brief Description

Route Critical Event allows the platform to support critical events by ensuring that these events are always captured and processed by the subscribing applications and services, and that the event is appropriately applied to settlement either as a settlement fund, settlement adjustment, or fee item. This use-case helps to provide "guaranteed" delivery of critical events.

5.1.10.2 Pre-Conditions

- The Message Broker has a "dead" letter exchange and process defined for events that either cannot be routed by the Message Broker or that cannot be processed by the subscriber due to an event format or data error
- The Dead Letter Exchange categorizes dead messages into the following:
  - Routing errors
    - The Message Broker cannot deliver the message due to an error in routing information
  - Delivery (Format or data) errors (Poison Messages)
    - The Subscriber cannot process the message due to an error in the message format or data
- Application logging in the canonical format is enabled for components involved in handling the critical event

5.1.10.3 Interaction Diagram



5.1.10.4 Flow of Events

1. A Tenant Customer creates a disputes chargeback in the Customer Portal.
2. The Customer Portal invokes a Dispute Action Service for the dispute chargeback.

3. Dispute Services processes the chargeback, logs the event, and makes a request of the Dispute Publisher Adapter to publish a Chargeback Event on the Message Broker.
4. The Dispute Publisher Adapter publishes the Chargeback Event to the Message Broker and logs the event.
5. If the Message Broker cannot deliver the event then the Message Broker sends the event to the Dead Letter Exchange for logging and handling by the Platform Operations group, logs the error event, and the use case stops.
6. If the Message Broker can deliver the event then the event is delivered to all subscribers, just the Settlement Adapter in this example and logs the event.
7. The Settlement Adapter validates the format and data of the event to make sure it can process the event.
8. If the Settlement Adapter determines there are no format or data errors then the settlement event is processed as a settlement adjustment in this example, placed into the proper format for the PULSE Settlement System, transmitted to the PULSE Settlement System for settlement processing, and then logged.
9. If the Settlement Adapter determines there are format and/or data errors then the settlement event is logged as an error and sent to the Message Broker's Dead Letter Channel for logging and handling by the Platform Operations group.

5.1.10.5 Post Conditions

- Critical events have either been successfully delivered and processed by all subscribers or the event is managed by the Dead Letter Exchange
- Critical Events have been logged by each component that has handled the critical event

5.1.10.6 Notes

1. Critical events must be logged by each component that touches the event with the unique application, service, or component ID to ensure that the critical event can be properly reconciled in case the event is not properly processed and does not arrive in the Dead Letter Exchange.
2. The ability to replay an event that has not been routed properly to all subscribers or has not been processed properly by all subscribers could require very complex application logic and this needs to be thought through as to the simplest and most effective solution, also the ability to correct routing logic and edit message formats and contents may not reside within the Operations Portal but may be a development task.
3. An alternative to replay may be a manual adjustment into the affected system, essentially a new manually created event type.
4. The Message Broker must log the publisher and all subscribers, as well as the date and time when an event is logged by the broker. This is required because changes to publishers and subscribers are dynamic and the logging of the event occurs within a specific point in time.

5.1.11 Upgrade Micro-Service Minor Version

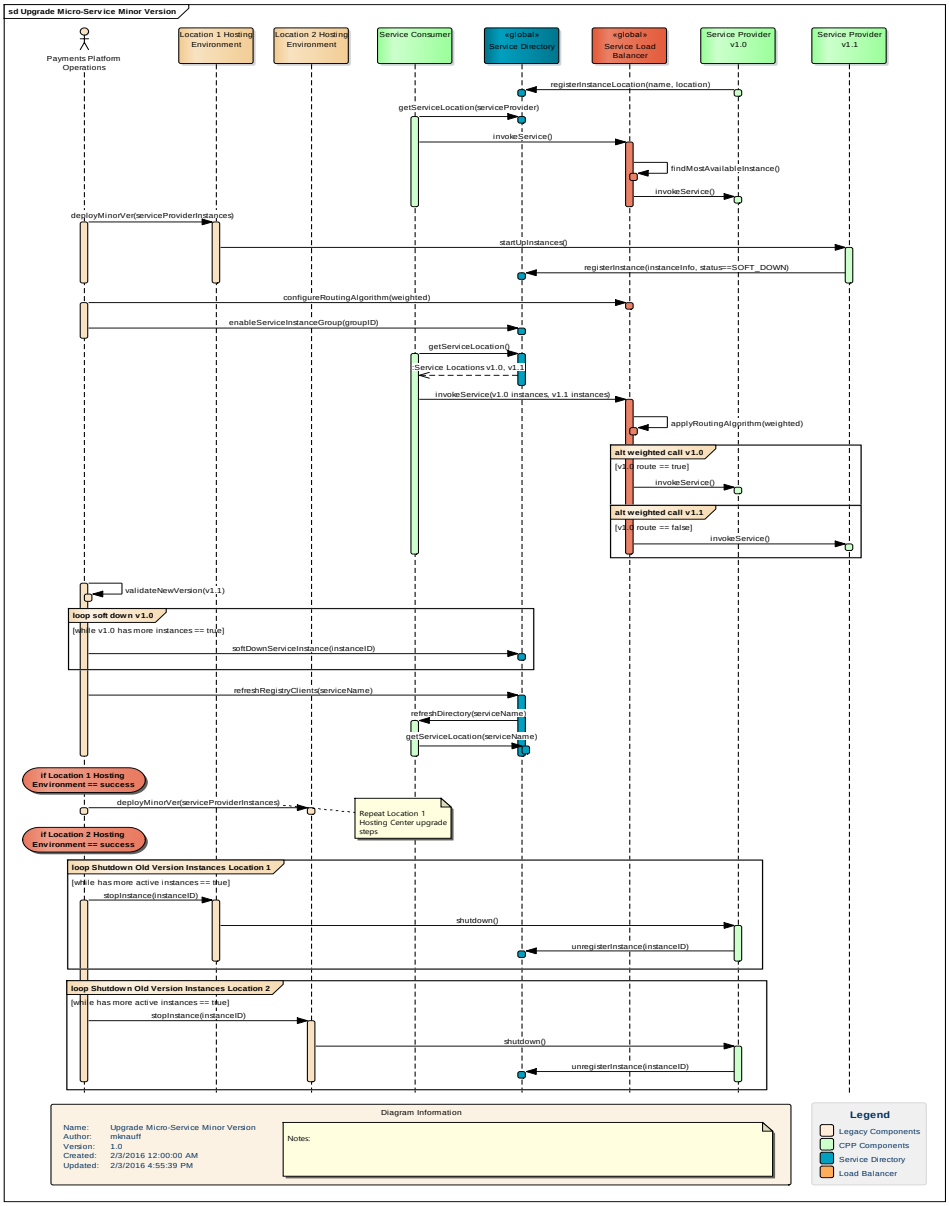
5.1.11.1 Brief Description

Upgrade Micro-Service Minor Version provides for the ability to modify and evolve services that don't affect their APIs with minimal impact to service consumers.

5.1.11.2 Pre-Conditions

- Service supports a method to identify minor versions of the service which is not part of API
- Service infrastructure can concurrently support two (2) production minor versions of the same service
  - Same API Interface
  - Service Logic may be different
  - Database or Data Repository may be different
- Service Registry supports multiple minor versions of the same service
- Service Registry can support "soft downing" of service instances
- Load Balancer can route service requests away from instances in a *soft down* state
- Consumers of the service are using the previous or original version
- There are at least two (2) environments hosting the service to be upgraded

5.1.11.3 Interaction Diagram



5.1.11.4 Flow of Events

1. Payment Platform Operations deploys a new minor version of the service into Location 1 of the Hosting Environment.
2. The Location 1 Hosting Environment sends a message to the new minor version instances (v1.1) to start.
3. Each new minor version Service (v1.1) instance registers itself with the Service Registry in SOFT_DOWN state so that the service instances are running but new service requests cannot be routed to them.
4. Payment Platform Operations configures the Load Balancers with a weighted algorithm (existing versus new version) for the service.
5. Payment Platform Operations sends a request to the Service Directory to enable all instances of the new minor version (v1.1)
6. Service Consumers make new requests for the Service Location of the upgraded service and the Service Directory returns all available instances of the Service including version 1.0 and version 1.1 instances.
7. The Service Consumer invokes the service via the Load Balancer and provides all available instances of the Service Provider to the Load Balancer.
8. The Load Balancer dispatches the service request to the appropriate Service Instance v1.0 or v1.1 according to the weighted routing algorithm.
9. Payment Platform Operations validates that the new version (1.1) is working correctly and gradually shifts more of the service requests to the v1.1 instances.
10. After all of the service requests are being routed to only version 1.1 service instances then Payment Platform Operations sends a message to soft down all version 1.0 instances of the service.
11. Payment Platform Operations sends a message to the Service Directory to refresh all clients of the service.
12. The Service Directory makes a callback to all registered clients of the service to refresh the locations of the service instances.
13. The Service Directory only returns version 1.1 service instances since all version 1.0 instances are in a SOFT_DOWN state.
14. The same process is repeated for the service running in Location 2 Hosting Environment.
15. After both Location Hosting Environments (1 and 2) have been successfully upgraded then Platform Operations sends requests in both Hosting Environments to stop the running version 1.0 instances.
16. The Hosting Environments send shut-down requests to each running version 1.0 service instance.
17. Each version 1.1 service instance unregisters itself with the Service Directory before terminating.

5.1.11.5 Post Conditions

- All service instances have been successfully upgraded to the new version (1.1) in each of the Hosting Environments (1 & 2)
- All old service instances (v1.0) have been successfully shut-down and removed from the Service Directory in each of the Hosting Environments (1 & 2)
- Consumers of the service are successfully utilizing the new service version (v1.1) and were not affected by the change in minor versions
- The service was available during the entire upgrade cycle

5.1.11.6 Notes

1. Service upgrades will not cause a service outage
    - Services will be continuously available during upgrades
    - Rolling upgrades of service will need to be supported either within a data center and/or across data centers
        - Blue-Green upgrades
        - Canary-in-a-Coal Mind upgrades
2. Changes to services will maintain backward compatibility whenever possible in order to avoid supporting multiple versions of the same service.
3. Versioning of a service will occur if backward compatibility cannot be maintained.
4. Versioning of a service if required will be temporary in order to provide clients with sufficient time to upgrade. Older versions will be deprecated and provided with a sunset date.
5. Services will support major and minor versions
    - Major versions indicate changes in the service interface (API)
        - Clients will need to be aware of these changes and implement changes of their own to accommodate the changes
    - Minor versions indicate changes that do not affect the API interface
        - Clients will not be aware of these changes
6. Service versions supported will include the following:
    - -1 = The previous version of the current production version
    - 0 = The current production version
    - +1 = The version currently under development (production candidate)
7. How will new versions of the canonical model be handled in Hbase and in Oracle?
    - Does each JSON object or row of data need to indicate a version number?
    - Can clients request specific versions of the data?
    - Do all data changes need to be backward compatible?

5.1.12 Upgrade Micro-Service Major Version

5.1.12.1 Brief Description

Upgrade Micro-Service Major Version provides for the ability to modify and evolve services that affect their APIs and impact service consumers.

5.1.12.2 Pre-Conditions

- Service supports major version designation as part of API
- Service infrastructure can support three (3) concurrent production versions of the same API
  - API Interface
  - Service Logic
  - Database or Data Repository
- Service Registry supports multiple versions of the same service
- Service Registry can support "soft downing" of service instances
- Load Balancer can route service requests away from instances in a *soft down* state

5.1.12.3 Interaction Diagram

5.1.12.4 Flow of Events

5.1.12.5 Post Conditions

TODO: Storage by client of by href of resources managed by the service provider?

5.1.12.6 Notes

1. Service upgrades will not cause a service outage
   - Services will be continuously available during upgrades
   - Rolling upgrades of service will need to be supported either within a data center and/or across data centers
     - Blue-Green upgrades
     - Canary-in-a-Coal Mind upgrades
2. Changes to services will maintain backward compatibility whenever possible in order to avoid supporting multiple versions of the same service.
3. Versioning of a service will occur if backward compatibility cannot be maintained.
4. Versioning of a service if required will be temporary in order to provide clients with sufficient time to upgrade. Older versions will be deprecated and provided with a sunset date.
5. Services will support major and minor versions
   - Major versions indicate changes in the service interface (API)
     - Clients will need to be aware of these changes and implement changes of their own to accommodate the changes
   - Minor versions indicate changes that do not affect the API interface
     - Clients will not be aware of these changes

6. Service versions supported will include the following:
   - -1 = The previous version of the current production version
   - 0 = The current production version
   - +1 = The version currently under development (production candidate)

5.1.13 View Application or Service Status

5.1.13.1 Brief Description

5.1.13.2 Pre-Conditions

5.1.13.3 Interaction Diagram

5.1.13.4 Flow of Events

5.1.13.5 Post Conditions

5.1.13.6 Notes

6. Logical View

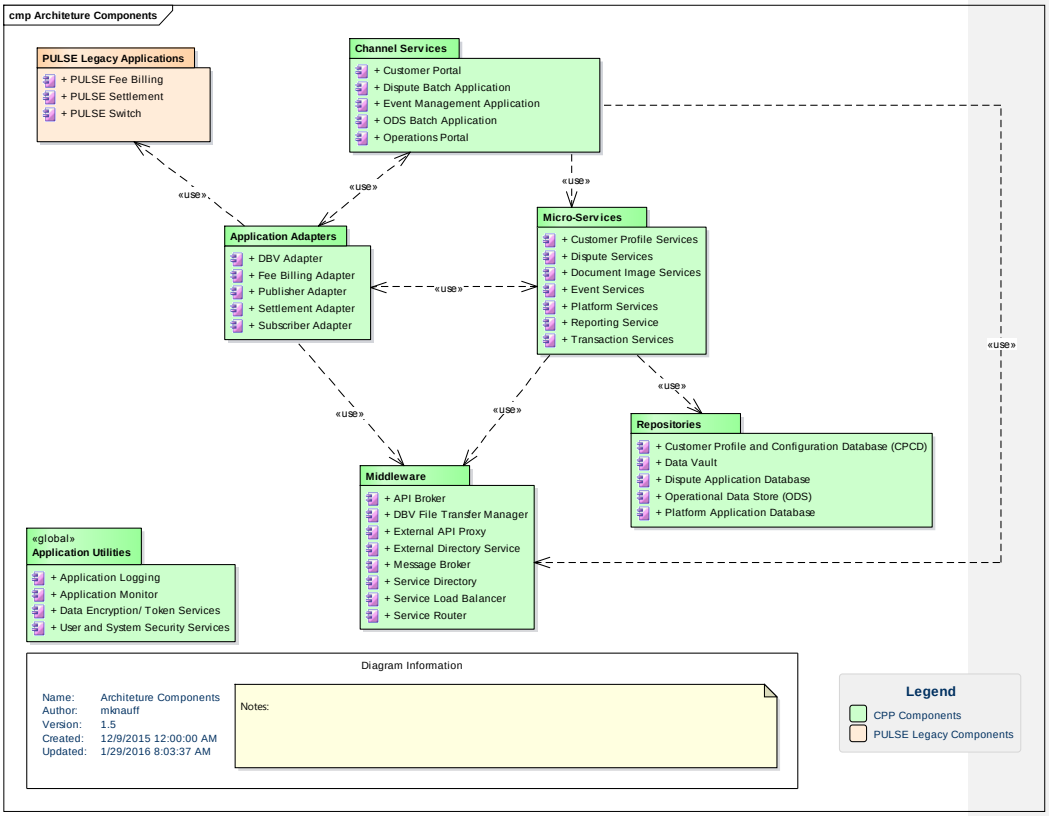


Figure 3 - CPP Disputes/PULSE Logical Architecture

6.1 Description

Platform components are organized into the following categories with each category having constraints on how they interact or use other platform component categories. These constraints are defined in order to limit dependencies and coupling between platform components. The restriction on dependencies between component categories helps to facilitate reuse and limits the impact of changes across components.

6.1.1 Channel Services

The *Channel Services* are a CPP category of components that provide user or batch interfaces to customers and/or that aggregate a group of related services under a single name space. The *Channel Services* are the top-level consumers of services.

6.1.2 Application Adapters

The *Application Adapters* are components that connect the CPP and PULSE Legacy applications and Micro-Services to middleware, primarily to the Message Broker. Adapters are specific to applications or services and middleware. A given adapter understands the communication protocol and data formats used by particular application or service, as well as the communication protocols and canonical formats used by the middleware such as the Message Broker. The purpose of the adapter is to isolate the technical details of the middleware from the application or service. The application or service communicates with the adapter in its own proprietary communication protocol and data formats. This causes Application Adapters and Applications or Services to be tightly coupled, which is why adapters and applications or services have bi-directional dependencies. Legacy Applications such as the PULSE Legacy Applications do not have a dependency on the Application Adapter in order to prevent Legacy Applications from having to implement CPP-specific changes.

6.1.3 Application Utilities

Application Utilities are components or services used generically by many CPP components, are business domain independent, and that are generally managed and supported at the enterprise level. These include logging, monitoring, and security services.

6.1.4 Micro-Services

Micro-Services are components that contain groups of closely related operations that manage a single bounded-context (domain) of data such as transactions, or dispute cases, or customer profiles, or events, etc. The *Micro-Services* are usually implemented as REST-ful (HTTP) services.

6.1.5 Middleware

Middleware are components that are responsible for brokering communication with other CPP or legacy components such as via APIs, file transfers, service directories, message brokers, load balancers, etc. The *Middleware* components are low level components that have no dependencies on the components that utilize them. *Middleware* may be utilized by *Applications*, *Micro-Services*, and *Application Adapters*.

6.1.6 PULSE Legacy Applications

The *PULSE Legacy Applications* are components that existed prior to the development of the CPP and are vendor developed and supported products. These applications have no direct dependency of the CPP components and are integrated into the CPP via Application Adapters that provide the communication protocol and data format conversions between the *PULSE Legacy Applications* and the CPP components.

6.1.7 Repositories

Repositories are the components for holding and managing the persistent storage of CPP data. Each repository contains the domain data for a single micro-service and repositories are paired with a micro-service that manages the interaction of other components with the domain repository. Only the micro-service responsible for the data domain is allowed to directly access the repository, all other components utilize the micro-service to interact with the domain data.

6.2 System Context

Commented [MJK11]: Correct PULSE Settlement from raw to setteled transactions

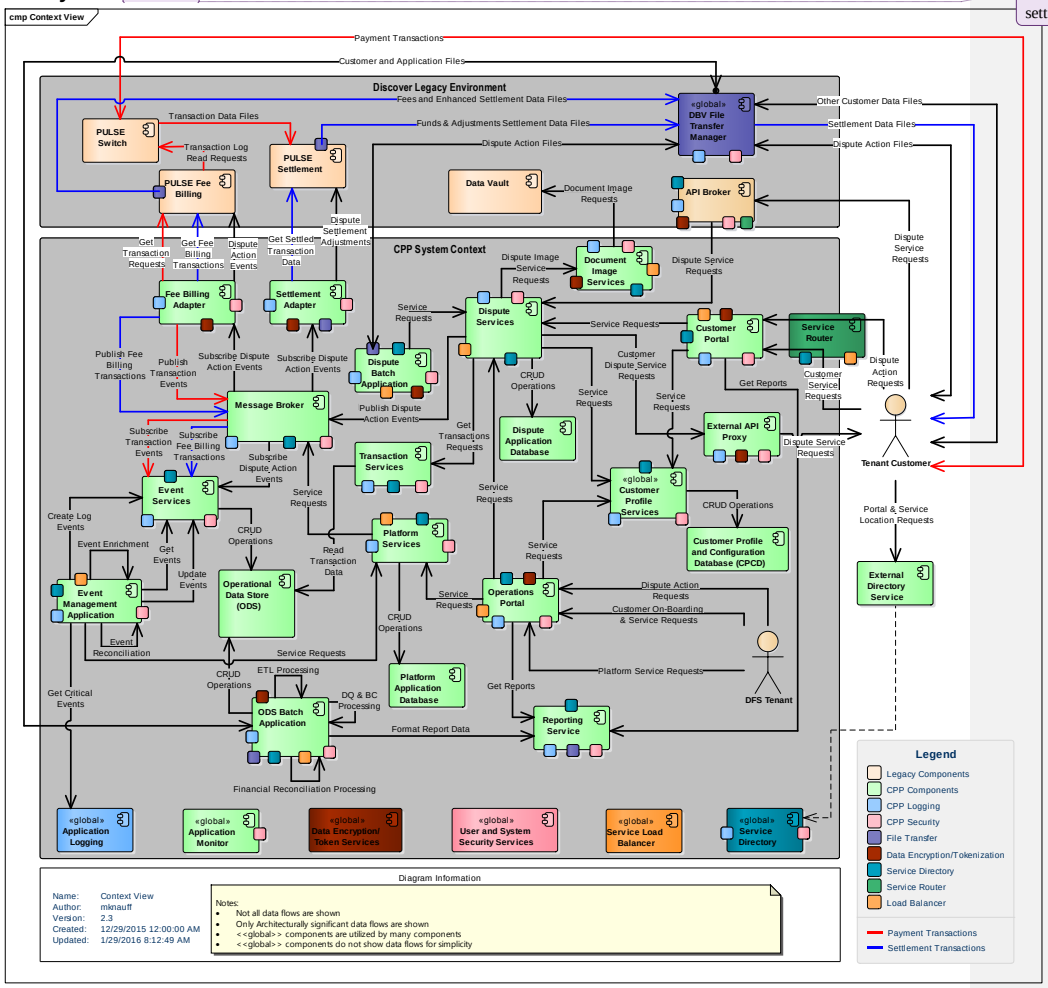


Figure 4 - CPP System Context Diagram

6.2.1 *Component Catalog*

6.2.1.1 API Broker

The *API Broker* is CPP component that provides access to CPP APIs to Tenant Customers. The *API Broker* provides API security, auditing, governance, and an external developer portal. All CPP services invoked through the *API Broker* are forwarded to an internal application services proxy that tokenizes PCI-sensitive data before forwarding the request on to the actual application service and then de-tokenizes the data before sending the response back to the customer.

6.2.1.2 Application Logging

The *Application Logging* is CPP component that is responsible for aggregating the logs produced by all CPP applications and services and placing these into a centralized repository for convenient access and monitoring of all CPP applications and services. The application and service log data is in a canonical format that can be easily ingested by other systems and processes. The three (3) uses of the log data are:

- Monitoring operational output of the internal processing state of the components
- Security auditing of the internal processing state and what user or processes are performing the requests
- Monitoring and reconciliation of critical business events such as financial events

6.2.1.3 Application Monitor

The *Application Monitor* is a CPP component that is responsible for monitoring the application and service log data and polling service and system KPI's and displaying these to interested parties through a variety of dash boards and reports via the Operations Portal.

6.2.1.4 Customer Portal

The *Customer Portal* is a CPP component that provides the user interface for Tenant Customers of the platform access to platform services such as disputes. Dispute related functions include the ability to create, view, and update dispute and adjustment cases as well as dispute and adjustment related reports. The *Customer Portal* also provides customer self-servicing functions and reports.

6.2.1.5 Customer Profile and Configuration Database

The *Customer Profile and Configuration Database (CPCD)* is a CPP component that provides the central repository and system of record for customer profile-related data. The *CPCD* is accessed via the Customer Profile Services micro-service, which encapsulates the data.

6.2.1.6 Customer Profile Services

Customer Profile Services is a CPP component that is responsible for providing access (create, read, update, and delete (expire)) to the *CPCD* data. The *Customer Profile Services* component encapsulates the technologies and formats utilized by the CPCD repository and enforce the business rules surrounding access to this data.

6.2.1.7 Data Encryption/Token Services

The *Data Encryption/Token Services* is a CPP component that is responsible for tokenizing all PCI-sensitive data within a semi-trusted security zone before applications send the data into a trusted security zone, which does not provide extensive PCI hardening. CPP Applications call the service to tokenize the data before sending it as part of service requests into trusted security zone service providers. CPP applications also call this service to de-tokenize the data before sending the data to external customers or systems that require the original PCI-sensitive data in the clear.

6.2.1.8 Data Vault

The *Data Vault* is largely Document Image Repository that provides for the storage, search, and retrieval of data-sensitive document images. Images that contain sensitive information are encrypted within the *Data Vault*. Other types of data that cannot be tokenized may also be placed within the vault in an encrypted format.

6.2.1.9 DBV File Transfer

The *DBV File Transfer* is a CPP component that provides the store and forward environment for the exchange of application and customer file between internal applications and between internal applications and Tenant Customers.

6.2.1.10 Dispute Application Database

The *Dispute Application Database* is a CPP component that provides the persistent repository for dispute case data. The *Dispute Application Database* provides the domain of data for the disputes micro-service. Access to the *Dispute Application Database* only occurs through Dispute Services.

6.2.1.11 Dispute Batch Application

The *Dispute Batch Application* is a CPP component responsible for processing dispute service requests submitted or sent by external customers in a batch file via the file transfer channel managed by the DBV File Transfer Manager. The *Dispute Batch Application* utilizes the Dispute Services for invoking the appropriate operations to process batch requests.

6.2.1.12 Dispute Services

Dispute Services is a CPP micro-service component that provides the ability for Tenant Customers to create, read, and update dispute and adjustment case data, as well as generate dispute and adjustment action events such as chargeback adjustments and ticket retrievals.

6.2.1.13 Document Image Services

Document Images Services is a CPP micro-service component responsible for storing, searching, retrieving as well as encrypting and decrypting sensitive document images. The *Document Inage Service* utilizes the Data Vault as its repository for data-sensitive document images.

6.2.1.14 Event Management Application

The *Event Management Application* is a CPP component responsible for aggregating all critical events on the CPP from the Application Logging service and ensuring that these are placed into the ODS via Event Services, as well as reconciling all critical events such as financial events.

6.2.1.15 Event Services

The *Event Services* is a CPP component that is responsible for subscribing to all events published on the Message Broker and cataloging them into the ODS. *Event Services* is also responsible fior providing access to events that have been placed into the ODS.

6.2.1.16 External API Proxy

The *External API Proxy* is a CPP component that provides an added layer of security for internal CPP applications and services that need to call services implemented by parties external to the CPP environment. An Example of this type of use case is Dispute Services implemented by Tenant Customers for dispute processing.

6.2.1.17 External Service Directory

The *External Service Directory* is a CPP component that allows Tenant Customers to resolve CPP service and application names with an available geographic location and address for the application or service.

6.2.1.18 Fee Billing Adapter

The CPP *Fee Billing Adapter* is responsible for converting Dispute Action Events from the Message Broker into the required communication protocols and formats for the PULSE Concourse System in order to apply fees and fee settlement to these events.

The CPP *Fee Billing Adapter* is also responsible for retrieving transaction events that are sent to the PULSE Fee Billing system from the PULSE Switch and publishing these to the Message Broker.

6.2.1.19 Message Broker

The *Message Broker* provides the highly-available and scalable messaging infrastructure for asynchronous communication between CPP components and also with the PULSE Legacy Applications. The *Message Broker* provides the ability for CPP components to publish events for interested subscribers and to subscribe to events that are generated by other components. The *Message Broker* is the connector for publish-subscribe events.

6.2.1.20 ODS Batch Application

The *ODS Batch Application* is CPP component that is responsible for accepting inbound data feeds from various CPP components and performing the required ETL processes for data that is being moved into or out of the ODS. The *ODS Batch Application* accepts data feeds from CPP components for loading into the ODS in order to make this data available to other systems or for reconciliation processing (Lambda Architecture Support: slow arriving data). The *ODS Batch Application* also provides data feeds to other systems that require ODS data in a batch mode including for the purposes of generating reports.

The *ODS Batch Application* is also responsible for accepting inbound data feeds from Tenant Customers and performing the required ETL processes for data that is being moved into or out of the ODS. The *ODS Batch Application* also provides data feeds to Tenant Customers that require ODS data in a batch mode for their own internal processing and/or reporting.

6.2.1.21 Operational Data Store

The *Operational Data Store (ODS)* is a CPP component that provides the central repository for the customer tenant transaction and event data. The *ODS* stores all event data in a canonical format, provides balance and control processes to provide for the System of Truth capabilities for this data, and stores history of both detail and summary-level data. The *ODS* is composed of multiple domains of read-only data including events, transactions, disputes, settlement, fees, etc.

6.2.1.22 Operations Portal

The *Operations Portal* is a CPP component that provides the user interface for Tenant Operations to create, view, update, and delete (expire) customer configuration and profile data. The *Operations Portal* also provides the Operations users with the access view platform services such as disputes, as well as CPP operational and platform data.

6.2.1.23 Platform Application Database

The *Platform Application Database* is a CPP component that provides the persistent repository for data related to the operations of the CPP as technology platform. The *Platform Application Database* provides the domain of data for the platform micro-service. Access to the *Platform Application Database* only occurs through Dispute Services.

6.2.1.24 Platform Services

Platform Services is a CPP component that is responsible for providing access to platform operational services such as the Message Broker event routing registry, event reconciliation data, replay of events, etc.

6.2.1.25 PULSE Fee Billing

The *PULSE Fee Billing* is the legacy system provided by the vendor BHMI and the Concourse product that is responsible for calculating transaction and disputes-related fees for PULSE.

6.2.1.26 PULSE Settlement

The *PULSE Settlement* system is an existing PUSLE legacy system that provides the calculation of settlement amounts between PULSE acquirers and issuers. The software for this system is provided by the vendor FIS and the CONNEX Advantage product for zOS (mainframe) platforms.

6.2.1.27 PULSE Switch

The *PULSE Switch* is the PULSE legacy system that is responsible for routing ATM cash and merchant POS transactions represented as ISO 8583 messages between acquirers and issuers (PULSE financial institutions) and/or their processors. The *PULSE Switch* captures these as financial transactions for settlement of fees and funds between parties. The software for this platform is provided by the vendor FIS and the CONNEX product.

6.2.1.28 Reporting Service

The *Reporting Service* is a CPP component that is responsible for formatting application data into human-readable format for display in various viewers or for printing as a hard copy of the report. The *Reporting Service* primarily utilized by the ODS to produce operational and customer reports.

6.2.1.29 Service Directory

The *Service Directory* is a CPP component that provides for service discovery and service name resolution. The *Service Directory* provides for loose coupling between services by providing resolution between logical names and physical locations of the service. CPP micro-services register themselves with the service in order to provide dynamic discovery of available instances of the service.

6.2.1.30 Service Load Balancer

The *Service Load Balancer* is a CPP software component responsible for knowing which service instances are most available and then distributing service requests to available instances based upon a specified algorithm.

6.2.1.31 Service Router

The *Service Router* is a CPP edge component that is responsible for providing content-based routing of CPP application or service requests on the edge or border of the CPP context. The *Service Router* allows routing rules to be defined based upon data values found within attributes of the requests coming from external platform parties. The *Service Router* works closely with the Service Directory and Load Balancer to route the service request to the appropriate component instance based upon the rules contained within the *Service Router*.

6.2.1.32 Settlement Adapter

The Settlement Adapter is responsible for converting Dispute Settlement Adjustements received from the Message Broker into the required communication protocol and formats for the PULSE Settlement System.

The Settlement Adapter is also responsible for receiving a copy of the PULSE Settlement file outputs and sending these in the appropriate format to the DBV File Transfer Manager for consumption by the ODS via the ODS Batch Application for the reconciliation of Dispute settlement adjustments and PULSE settlement.

6.2.1.33 Transaction Services

Transaction Services is a CPP component that provides access to payment transactions stored within the ODS. *Transaction Services* is a micro-service that provides read access to a specific domain of data related to payment transactions that is stored within the ODS.

6.2.1.34 User and System Security Services

The *User and System Security Services* is a CPP component that provides user and system authentication, single sign-on support, as well as managing and providing authorization via access control lists.

6.3 System Data Flow

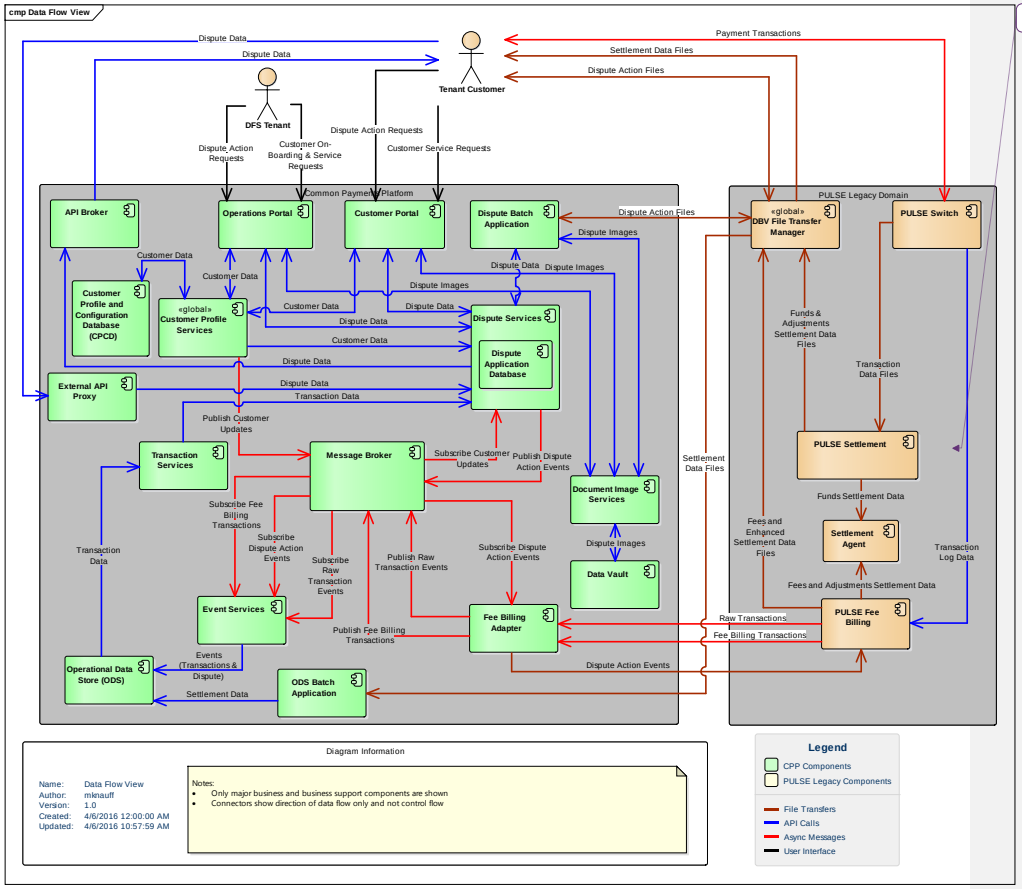


Figure 5 - CPP Logical Data Flow

6.3.1 *Data Channel Types*

6.3.1.1 Customer Data

6.3.1.2 Dispute Data

6.3.1.3 Fee Billing Data

6.3.1.4 Settlement Data

6.3.1.5 Transaction Data

Formatted: Body Text, No page break before

Formatted: Font: 8 pt

Formatted: Heading 4, No page break before

Formatted: Body Text, No page break before

6.4 CPP Service Standard Defined Form

All CPP services follow a standard define form in their architecture, design, and implementation. This includes the following:

6.4.1 Micro-Services Pattern

The following definition is implied for the description that follows:

- A *Service* is a collection of related operations. Each operation has a corresponding API, which implies that a *Service* is realized through one (1) to many APIs.

The Micro-Services pattern is characterized by the bundling of like operations into a single service that can be developed, managed, and deployed as a cohesive group of APIs when they all can reference the same resource, or group of resources and data. This is the key attribute of micro-services and what distinguishes them from just a loose collection of services. A true micro-service must operate on the same resource or set of related resources in order to be defined as cohesive. The micro-service must either be very loosely coupled or not coupled all to other resources in order to maintain the integrity of the micro-service.

The set of related resources that a micro-service operates on is referred to as a bounded-context. A bounded-context is the set of related operations and resources from a business (functional) perspective. The purpose of this pattern is the realization of modularity for the following purposes:

- Independent development, maintenance, and testing of the service apart from other services, which realizes modifiability, maintainability and testability, and includes the ability to utilize the most appropriate environment and tooling for the context
- Deployment of the physical software service module and its dependent sub-components into independent or separate virtual machines apart from other services, which realizes horizontal scalability and also modifiability and maintainability

The Micro-Services Pattern is combined with the **Error! Reference source not found.** to provide vertical and horizontal partitioning of functions in order to realize loose coupling and modifiability. The Micro-Services Pattern provides vertical segmentation from a business capability perspective, and the **Error! Reference source not found.** provides horizontal segmentation and abstraction from a technical or infrastructure perspective. See **Figure 6 - Micro-Services Logical View**

Formatted: Font: Bold, Font color: Blue

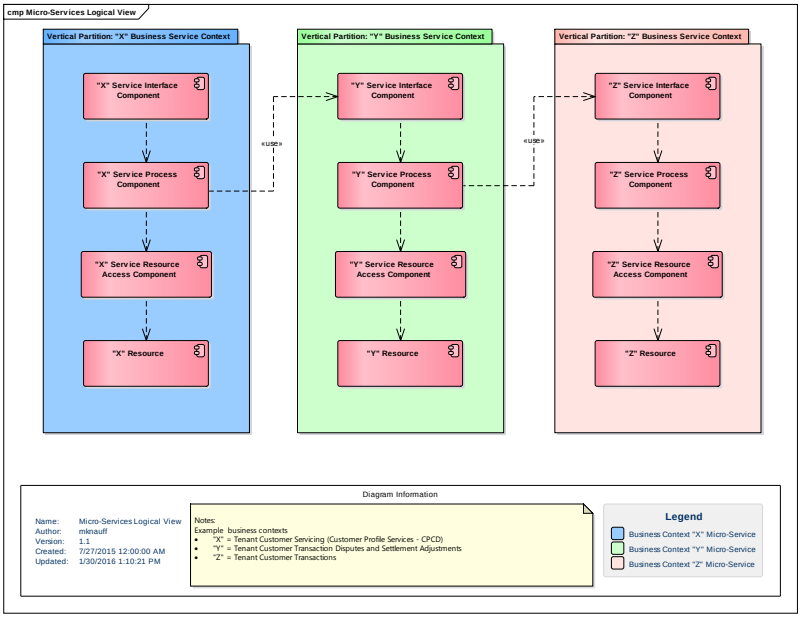


Figure 6 - Micro-Services Logical View

The vertical and horizontal segmentation provides great flexibility of development and deployment models and decouples these decisions from the architecture. The pattern lends itself well to both horizontal scalability and high-availability when resources required by the micro-service can be independently deployed with each instance of the micro-service and the resource can support an eventual consistency model (See **Figure 7 - Micro-Services Deployment with Eventual Consistency of Shared Resource**).

Formatted: Font: Bold

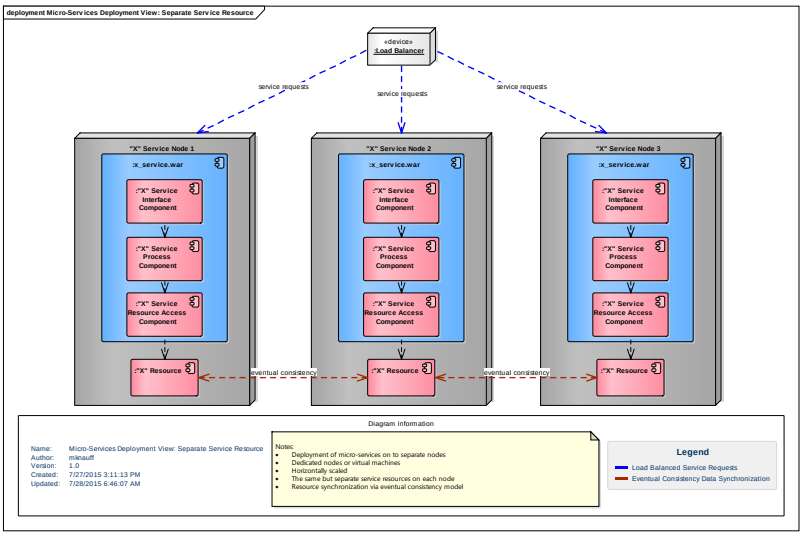


Figure 7 - Micro-Services Deployment with Eventual Consistency of Shared Resource

Many of the benefits of the micro-services model can still be realized in terms of flexibility of development and deployment models when a shared resource must be accessed by all instances of the micro-service for high consistency of the resource. The micro-service can still be horizontally scaled but this scalability is somewhat gated by the scalability of the shared resource. Availability of the micro-service also suffers from the coupling to the availability of the shared resource; however this cannot be avoided when the resource must be kept highly consistent (see **Figure 8 - Micro-Services Deployment with Highly Consistent Shared Resource**).

Formatted: Font: Bold, Font color: Blue

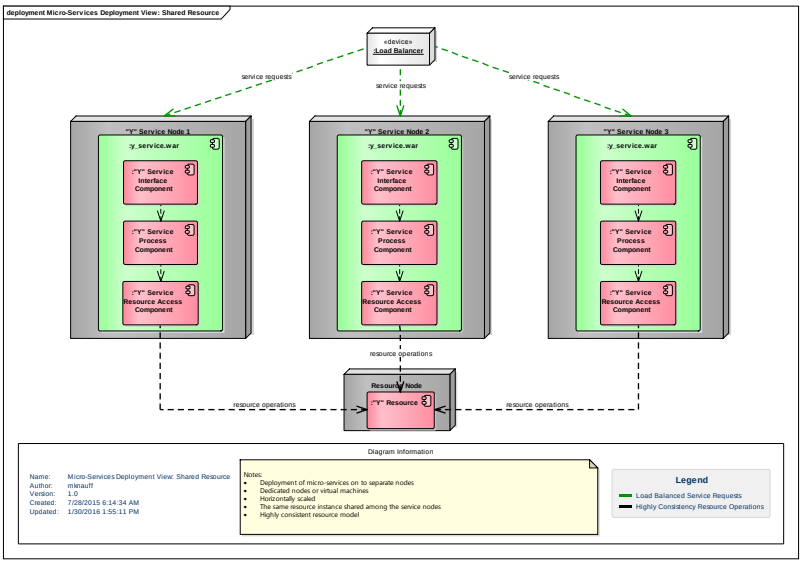


Figure 8 - Micro-Services Deployment with Highly Consistent Shared Resource

6.4.2 Stateless

CPP services shall be stateless in order to facilitate scalability and flexibility of the service. Client state shall not be represented within the service. If the service request depends on a specific client state then each service request shall include the client state as part of the request.

6.4.3 Layers Pattern

A service is implemented into four (4) specific application layers provide support for modularity and the isolation of changes within the application-specific code. These layers include the following:

6.4.3.1 Interface Layer

The *Interface Layer* contains all packages and classes associated with accepting user requests. These requests may be initiated by people via a user interface or via other systems via an API. This layer is the client-facing layer that implements portions of the application that are specific to the interface type of the client. Application components in this layer translate the service requests from the format that is specific to the interface channel and into the specific protocol and format that is specific to the application service.

The *Interface Layer* implements the *Front Controller Pattern* for handling and dispatching client requests. The *Interface Layer* may only communicate with the layer immediately below it, which is the …

6.4.3.2 Process Layer

The *Process Layer* contains all packages and classes associated with receiving requests from the *Interface Layer* and applying the processing or sequencing (orchestration) logic required to fulfill the request. This layer contains the Application Objects (AO) that implement the sequencing logic. The *AOs* implement a *Façade Pattern* for aggregating and simplifying the functionality of many Business Objects in the *Business Layer* that are required to execute the business logic.

6.4.3.3 Business Layer

The *Business Layer* contains all packages and classes that encapsulate the actual application logic for the business domain. This layer contains the Business Objects (BO) that are responsible for providing the required service data and data validation logic.

If the *Business Layer* requires access to other resources to process the request such as databases, files, or other system services then the *Business Layer* communicates with the layer immediately below it, which is the …

6.4.3.4 Resource Access Layer

The *Resource Access Layer* contains all packages and classes associated with calling other resources and managing their proprietary APIs and communication protocols. Requests for additional resources from the *Business Layer* are handled in this layer, which encapsulates the details of interacting with various resources outside of the application.

This layer contains the *Resource Entity Objects (REO)* that implement the orchestration, coordination, and sequencing steps required to retrieve and manipulate resources across several *Resource Access Objects (RAOs)*. The *RAOs* are responsible for providing the required communication protocol and data formats that are specific to a resource or service. The *REOs* implement the *Façade Pattern* for this layer. (See Figure 9 - Application Layers Pattern)

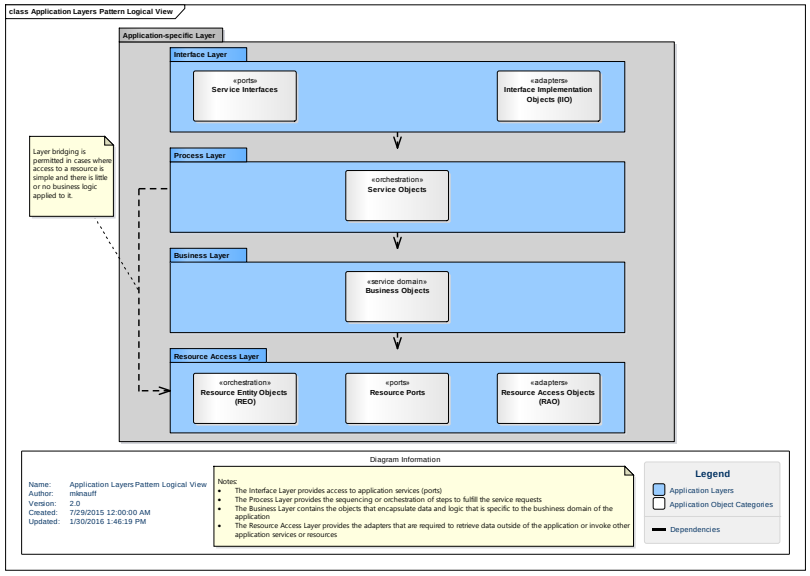


Figure 9 - Application Layers Pattern

6.4.4 Standard Interfaces (Ports and Adapters)

The *Ports and Adapters* (Hexagonal) *Pattern* provides a more elegant means for applications to deal with multiple client and interface types invoking the application services, as well as the need for the application to invoke multiple service providers that require a variety of communication protocols and interface types. The traditional Layers Pattern provides a good solution for applications that require support for only one (1) interface type to invoke the application service and have only one (1) service provider in the form of a database.

Newer applications have more challenging requirements in the form of the number of client types that invoke their services and the number and types of resources that the application must utilize to deliver services. The number and types of service consumers of the application as well as the number and types of service providers to the application also presents challenges in testing the application. The application behavior must be validated in isolation from the service providers and service consumers as well as in various stages of integration with the service consumers and service providers. The ability to mock the various service providers and service consumers is an important requirement for testability of the application or component.

The *Ports and Adapters Pattern* provides a modification to the Layers Pattern that allows for the application or component to elegantly support multiple service consumer types, multiple service provider types, and increased testability. The pattern defines various *Port* types, which provide specific types of capabilities. Typical *Port* types that support service consumers include:

- Action Port – invoke the primary application services or use-cases
- Trigger Port – notifies the application of an internal or external event

Other *Port* types may be supported depending on the nature of the application. Each *Port* type supports a standard interface and message type, the canonical form of the *Port*. Each *Port* or interface can also support multiple interface *Adapter* types. The *Adapters* support the specific communication protocol and message format type of the service consumer, and performs the conversion from the communication protocol and message format of the service consumer into the canonical form of the *Port*. Adapters are the physical implementation of the *Port/Interface*. This approach allows multiple consumer types to be supported including real time, batch, test mock, etc.

The application can also support multiple service provider types as well through the use of *Ports*. An internal *Port* or interface is created for each service provider type. Typical service provider types include:

- Repositories – invoke other component or application services or database services for resources
- Notifications – send notifications to other components or applications that an event has occurred within the application

Other *Port* types may be supported depending on the nature of the application, just like on the service consumer side. Each *Port* type supports a standard interface and message type, the canonical form of the *Port*. Each *Port* can also support multiple interface *Adapter* types. The *Adapters* support the specific communication protocol and message format type of the service provider, and performs the conversion from the canonical form of the *Port* to the communication protocol and message format of the service provider. Just like the *Adapters* for the service consumers, the *Adapters* for the service providers provide the physical implementation of the *Port* interface. The *Adapters* can be determined at runtime depending on the runtime environment requirements. This approach allows multiple service provider types to be supported including real time, asynchronous, batch, test mock, etc.

The modifications to the Layers Pattern (see **Error! Reference source not found.**) occur in the following ways:

- Interface Layer
  - Multiple *adapter* objects that can support various service consumer and interface types
    - The *adapter* objects support the client specific communication protocols and message formats, and perform the translation of the client-specific formats and into the standard format of the *port* type
  - Multiple *Port* type objects that support a canonical form of a specific functional aspect of the application or component
    - Each *port* type can support multiple *adapter* object types

- The appropriate *Adapter Object* can be specified at runtime
- Each *port* invokes the required *Application Object* to execute the client request

- Resource Access Layer
  - Multiple *Port* type objects that support a canonical form of specific service provider types
    - Each *port* type can support multiple *adapter* object types
    - The appropriate *Adapter Object* can be specified at runtime
    - Each *port* invokes the required adapter object to execute the service provider request
  - Multiple *adapter* objects that can support various service provider and interface types
  - The *adapter* objects support the service provider-specific communication protocols and message formats, and perform the translation of the standard format of the *port* type and into the service provider-specific formats and communication protocols

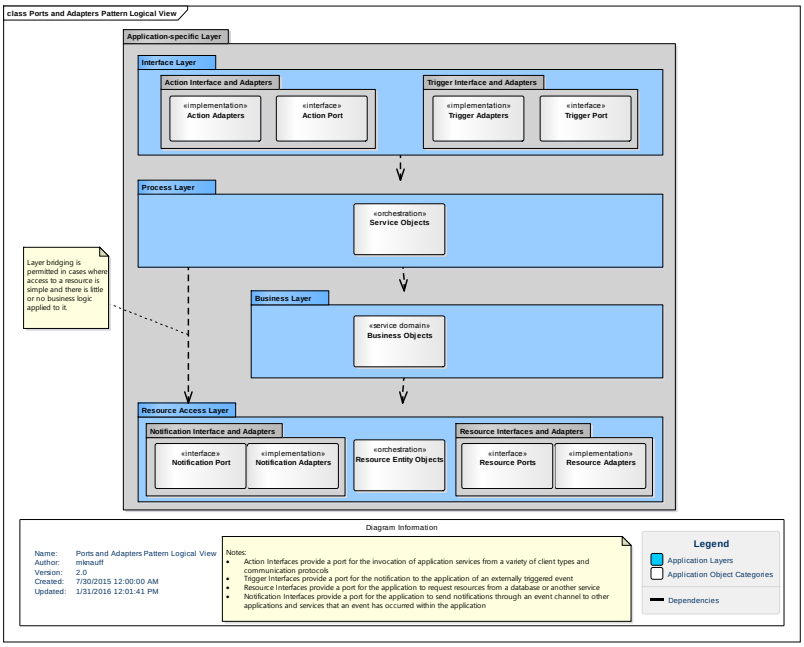


Figure 10 - Ports and Adapters Logical View

6.4.4.1 Standard CPP Ports and Adapters

A typical CPP Platform Micro-Service has a set of standard set of ports and adapters defined, although not every service may make use of all of them.

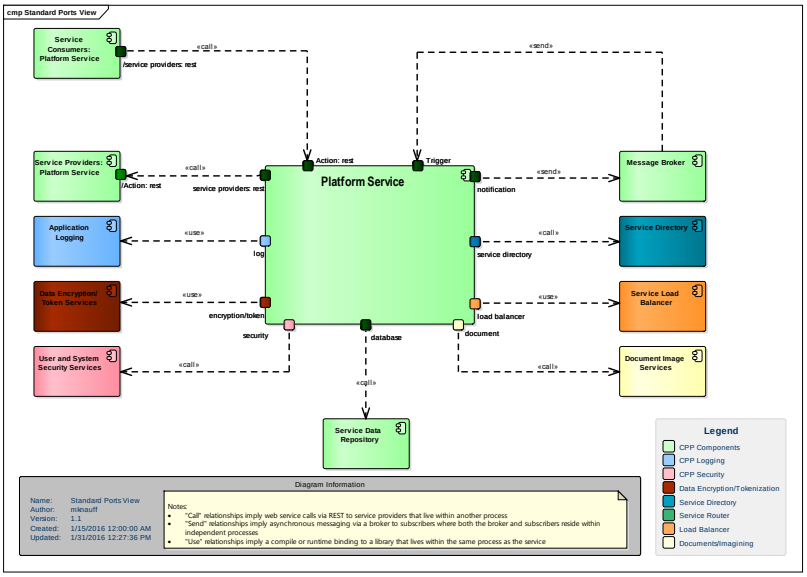


Figure 11 - Standard CPP Micro-Service Ports

- Service Consumer Ports
  - Action Port
    - REST Adapter
  - Trigger Port
    - JMS/AMQP Subscription Adapter
- Resource Ports
  - Application Logging Port
    - SLF4J/Log4J Adapter
  - CPP Micro-Service Resources (n)
    - Service-specific REST Adapter
    - Mock Adapter
  - Database Port
    - JPA Adapter
    - HBase Adapter
    - Mock Adapter
  - Data Encryption and Token Services Port
    - Protegrity Adapter
    - Mock Adapter

- Directory Services Port
  - Netflix Eureka Adapter
  - Mock Adapter
- Notification Port
  - JMS/AMQP Publisher Adapter
  - Mock Adapter
- Security Port
  - CPP Security REST Adapter
  - Mock Adapter
- Service Load Balancer Port
  - Netflix Ribbon Adapter
  - Mock Adapter

6.4.5 Common Technology Stack

7. Process View

[This section describes the system's decomposition into lightweight processes (single threads of control) and heavyweight processes (groupings of lightweight processes). Organize the section by groups of processes that communicate or interact. Describe the main modes of communication between processes, such as message passing, interrupts, and rendezvous.]

8. Deployment View

8.1 High-Level

Figure 12 - CPP Phase 1 Logical Deployment Architecture

8.1.1 Component Catalog

8.2 Security View

8.3 Availability View

9. Implementation View

[This section describes the overall structure of the implementation model, the decomposition of the software into layers and subsystems in the implementation model, and any architecturally significant components.]

9.1 Overview

[This subsection names and defines the various layers and their contents, the rules that govern the inclusion to a given layer, and the boundaries between layers. Include a component diagram that shows the relations between layers.]

9.2 Layers

[For each layer, include a subsection with its name, an enumeration of the subsystems located in the layer, and a component diagram.]

10. Data View

10.1 Multi-Tenant Domain Model

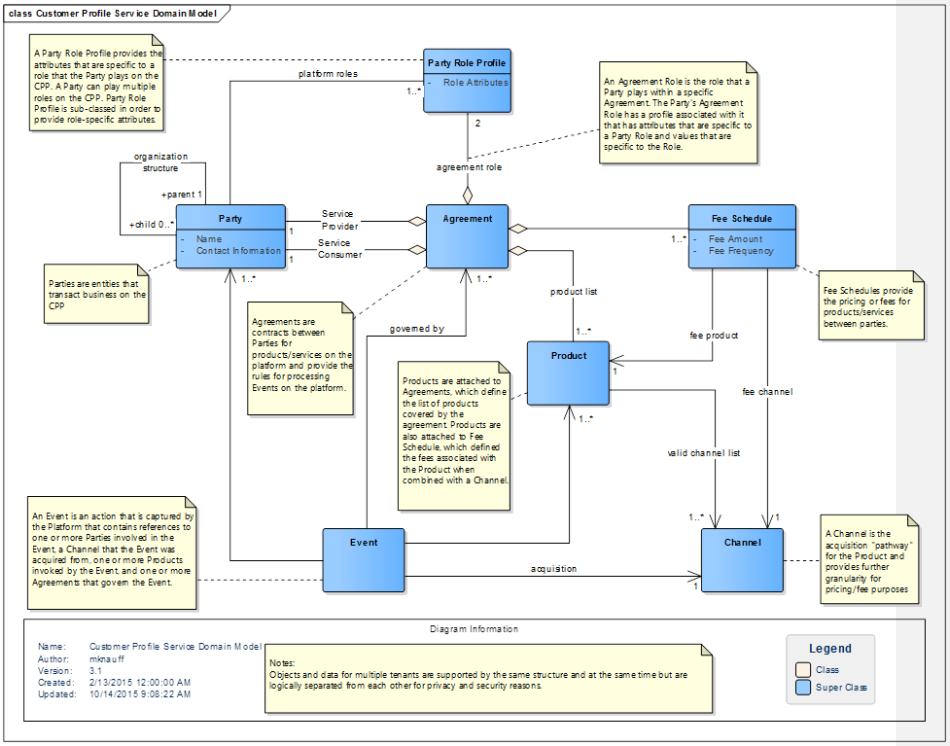


Figure 13 - CPP High-Level Domain Model for Multi-Tenancy

10.1.1 Agreement

An *Agreement* is a contract between the Platform and a Party or between two (2) Parties on the Platform. *Agreements* define the roles of each Party in the *Agreement* as well as the platform processing rules such as the Fee Schedule associated with each Product defined by the *Agreement*. Each *Agreement* defines a Service Provider that provides the Products on the Platform and the Service Consumer that utilizes the Products.

Examples:

- PULSE Issuer Agreement
- Diner Club International Franchise Agreement
- Discover Net-to-Net Agreement

10.1.2 Channel

A *Channel* is a pathway that enables Parties to exchange Events that are related to Products on the Platform. *Channels* also help to provide further granularity for fees that apply to a Product.

Examples:

- ATM Channel
- POS Channel
- Mobile Device Channel
- ecommerce Channel

10.1.3 Event

An *Event* is a meaningful action that occurs between Parties and that is captured by the Platform. An *Event* has a Product and a Channel associated with it. The *Event* is utilized by the Platform to determine the Agreement and Fee Schedule that govern an *Event*.

Examples:

- Sale Authorization Event
- Transaction Chargeback Event
- Funds Settlement Event
- Mobile Wallet Provisioning Event

10.1.4 Fee Schedule

An Agreement has one or more *Fee Schedules* associated with it. The *Fee Schedule* defines the fees that are applied to Products and Channels for Events that take place on the Platform.

Examples:

- PULSE Issuer ATM Fee Schedule
- Discover Consumer Rewards Credit Acquirer Interchange Fee

10.1.5 Party

A *Party* is entity (payment network, issuer, acquirer, merchant, processor, service provider (vendor), payment service provider, etc.) that is recognized by the Platform as a "customer" of the platform i.e., uses Platform services.

Examples:

- Apple
- Bank of America
- Diners Club Singapore
- Google
- PayPal
- Vantiv
- First Data Merchant Services
- PULSE Network

10.1.6 Party Role Profile

The *Party Role Profile* defines the set of attributes and their values that are specific to a role that the Party plays on the Platform. A Party can play many roles such as acquirer, acquirer processor, payment network, etc. as defined by separate Agreements.

Examples:

- Acquirer

- Acquirer Processor
- Buyer
- Digital Wallet Provider
- Issuer
- Issuer Processor
- Independent Sales Organization (ISO)
- Merchant
- Payment Network
- Supplier

10.1.7 Product

A *Product* is an offering provided by one Party (Service Provider) to another Party (Service Consumer) on the Platform. A *Product* is a service that is defined by an Agreement between the Parties and has a Fee Schedule associated with it.

Examples:

- PULSE ATM
- PULSE Pay
- Discover Consumer Rewards Credit
- Discover Debit

10.1.8 Service Consumer

A *Service Consumer* is a Party that utilizes a Product on the Platform.

Examples:

- Walmart
- JCB
- Discover Card

10.1.9 Service Provider

A *Service Provider* is a Party that provides a Product on the Platform.

Examples:

- Discover Network
- Diners Club International
- PULSE Network
- PayPal

11. Quality

[A description of how the software architecture contributes to all capabilities (other than functionality) of the system: extensibility, reliability, portability, and so on. If these characteristics have special significance, such as safety, security or privacy implications, they must be clearly delineated.]

11.1 Availability

11.2 Scalability and Performance

11.3 Modifiability

11.4 Security

11.5 Testability

12. Appendix A: Outstanding Design Issues

12.1 Outstanding Key Architecture/Design Issues

12.1.1 General

- ODS/Event technology stack and architecture
  - Scalability and performance
  - Support for Lambda requirements
  - Reconciliation/enrichment between fast and slow arriving data
  - Platform financial reconciliation processing
  - Event/messaging reconciliation
- Reporting Tool Selection
  - Assume Bert for now (OSS)
    - http://www.eclipse.org/birt/
- Security
  - Edge/Core Zone Pattern based upon data tokenization
  - User/system authentication, authorization, and access control
- Distributed transaction support
  - Micro-services ACID properties
  - Logical transaction scope and eventual consistency of transaction properties
    - "Business" transactions and event/message reconciliation processes
- PSEB architecture and implementation pattern(s)
  - Availability, scalability, and performance
    - How do we achieve high availability cross data center RabbitMQ clusters?
    - Can federated queues be used to connect RabbitMQ clusters across data centers?
  - Distribution of bus responsibilities across bus components
  - Application and service integration patterns
  - Physical deployment model
    - Are there any restrictions on the use of RabbitMQ configurations that span inside and outside PCF?
  - How does PCF achieve message persistence and consumer durability within RabbitMQ so that messages are guaranteed to be delivered at least once?
- CPP integration with Concourse
  - PULSE Switch Transaction Feed to PSEB/ODS (Lambda – Fast)
    - Partial transaction data
  - Dispute Fees Feed to Concourse
- CPP integration with IBM FIS CONNEX Advantage Settlement (PULSE Settlement)
  - PULSE Switch Transaction Feed to PSEB/ODS (Lambda – Slow)

Commented [MJK12]: Simon D.

Commented [MJK13]: Simon D.

Commented [MJK14]: Simon D.

Commented [MJK15]: Simon D.

        - Complete switch transaction data
    - Dispute Adjustment Feed to PULSE Settlement
    - PULSE Settlement Feed to PSEB/ODS
        - Disputable transactions
- Support for multiple service and application availability SLAs but no maintenance-induced outages
- Logging and monitoring technology stack and architecture
    - Proposed Enterprise solution is based upon ELK and for the PCF environment only
    - How will we handle aggregating logs outside of PCF (Splunk?)?
    - What will be centralized log repository and what tool will be used to search the repository?
    - Custom tags in the output of SLAM, for examlpe we could include request/correlation ids and log domains (AUDIT, APP LOG, PERF. MATRICS, ERROR) in the log output
        - Will ElasticSearch index on these tags?
        - What are these platform tags and will EA deliver a centralized logging capability that covers apps within and outside of PaaS ?
- Application reporting technology stack and responsibilities

12.1.2 PaaS/PCF

- PCF High-Availability Configuration
    - Cross-data center deployment
- PCF Managed Services High-Availability and Segmentation Configuration
    - Implementation for different application SLA categories
    - Isolation from the effects of other unrelated service clients
    - Cross-data center support
    - How does Siteminder SSO achieve high availability within PCF?
    - What are the plans for providing cross data center high availability production ready databases?
- Support for different service and application security zone segmentations
- CPP services high-availability design
    - Use of a Directory Service (Netflix Eureka)
    - Use of service health checks, failure detection, and recovery (Netflix Hystrix)
    - Use of Service or Application Client Load Balancers (Netflix Ribbon)
    - How do we configure external HTTP traffic to automatically load balance & failover to multi data center UI servers hosted within PCF?
    - We may want location affinity, where all traffic from external sources to UI, business tier & RabbitMQ servers within PCF all reside within the same data center as the database and automatically failover to another data center when the database is failed over. This is to reduce network latency. Is this achievable and how?
- Support for Batch Processing
- Use of Layer 7 for services within PCF

Commented [MJK16]: Simon D.

Commented [MJK17]: Simon D.

Commented [MJK18]: Simon D.

Commented [MJK19]: Simon D.

- Integration of non-PCF clients with services defined within PCF
- Integration of PCF clients with services or applications defined outside of PCF (user-defined services)
- Service for storing the passwords in an encrypted format (e.g. Vault Service) on PaaS Environment:
    - Pivotal Cloud Foundry encrypts service credentials at rest, although they are fully visible through dumping them via CLI or logging them. Pivotal is working on a solution that entails a separate Rest endpoint protected via an organization role that allows one to retrieve the credentials. Is there any update on when this solution will be available?

Commented [MJK20]: Simon D.

12.1.3 PULSE Concourse Interface

We now have more clarity on how we can publish Connex raw data and Concourse enriched normalized data to the ESB (Enterprise Service Bus).

- We have Concourse readers sitting on the HP-Non Stop (Connex). The Concourse readers read all the attributes related to the transaction from the Connex loggers.
    - The raw data with all attributes is made available to the application server. The writers are part of the application server. The writers can publish to the ESB data in raw format with all attributes before the data is parsed, normalized and enriched (Fees, product id etc.) by Concourse. This is how we can feed raw Switch data as-is to the BUS and eventually to ODS
- Once the raw data is made available to the application server, it is parsed, normalized, enriched (fees, product id etc.) and is published to the Concourse database. The writers can write this enriched data to ESB in a normalized format without lot of impact to the Concourse system efficiency. This way we need not to hit the Concourse database to get enriched data.
    - This is how we can feed enhanced/enriched (fees, product id etc.) data to the ESB and eventually to ODS
- The raw data and the enriched data can have same life cycle signature so that we can link them together, if needed within ODS.
    - A Life Cycle is the collection of all activity details from all external sources (including connex) related to a single cardholder initiated transaction.

Questions:

- We would need to know what normalized (enriched) data attributes we would need within ODS from Concourse and how we can write to the bus?
    - Mike K: We are going to want for sure all of the fee information associated with the transaction at a minimum. We will need to review the available enriched data attributes to see what we may want to capture. We are also going to want the data that is sent to PEP+ for ACH settlement and the detailed data that is sent to the FI's for their reconciliation processing for our own balancing and reconciliation processing for the fee data that is being published to the bus.
- Do we want Concourse to re-try publishing transactions (by Queuing) to the bus in case the bus is not available for some reason?
    - Mike K: Yes. The Bus will provide an ACK message when it has successfully received an event or message from a publisher. We will need to include George Harley in the conversations with BHMI at the appropriate time. George is considering the creation of a client-side library for the bus that could make BHMI's interface to our bus much simpler.