# Client Relationship Data Service (CRD) - Architectural Document

# 1. Executive Summary

This document outlines the architecture for the new Client Relationship Data Service (CRD). The CRD will replace existing platforms (B2C2 and Client Tiering System) and serve as a centralized system for client relationship data by being the golden source for client tiering data and the authoritive source for client mappings, product enablement, and coverage information.

**Key architectural decisions:**

* Microservices architecture using Spring Boot
* React-based web frontend.
* Microsoft SQL Server for persistent storage
* Redis for caching
* Kafka for event-driven communication
* FpML for standardized data exchange
* GraphQL API for complex, nested queries, complementing the primary REST API

**Benefits:**

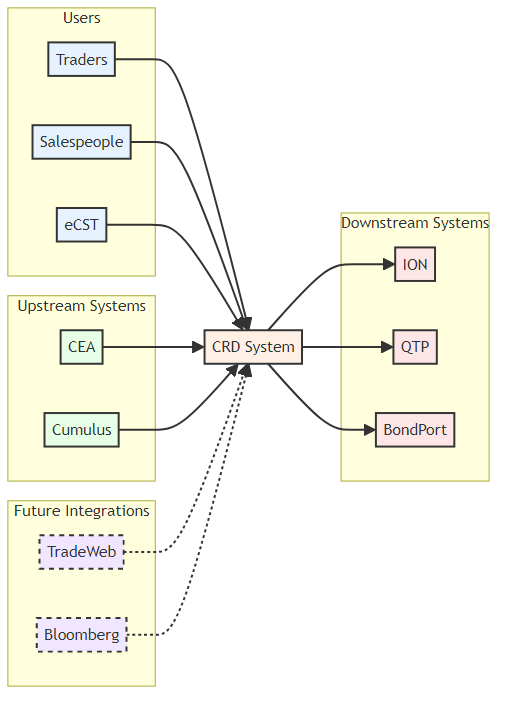
* Improved data consistency across systems
* Reduced manual interventions.
* Enhanced query capabilities for traders and salespeople - Centralized, authoritative source for client relationship data

**Challenges:**

* Complex integration with multiple upstream and downstream systems
* Data migration from legacy systems
* Ensuring high performance and scalability

# 2. System Context

The following diagram illustrates the high-level system context of the CRD, highlighting its interactions with various components:



**Users**: Our primary stakeholders - Traders, Salespeople, and the electronic Client Service Team (eCST).

**Upstream Systems**: Key data providers like CEA (Client Electronic Access) and Cumulus, which feed essential client information into the CRD.

**CRD System**: The core component that processes, stores, and distributes client relationship data.

**Downstream Systems**: Consumers of CRD data, including ION, QTP, and BondPort, which use this information for various business functions.

**Future Integrations**: Planned connections with systems like TradeWeb and Bloomberg, to enable seamless synchronization between CRD and venues.

## 2.1 Business Context and Drivers

The CRD project is driven by the need to:

* Consolidate client relationship data from multiple sources.
* Improve data consistency and reduce manual interventions.
* Provide a single, authoritative source for client mapping and tiering data.
* Enhance query capabilities for traders and salespeople.
* Streamline integration with downstream systems.

## 2.2 Stakeholders and Concerns

|  |  |
| --- | --- |
| Stakeholder | Concerns |
| Traders | • Require quick access to client enablement data  • Need to view client facing entity mapping data |
| Salespeople | • Need to view client tiering data  • Require information on client coverage and product tiering |
| Trading Systems Team | • Require reliable API for accessing client data  • Need consistent data format for integration |
| Algo Team | • Require programmatic access to client tiering data |
| IT Operations | • Concerned with system stability and performance  • Require monitoring and alerting capabilities |
| Compliance Team | • Need audit trails for data changes  • Require reporting capabilities for regulatory purposes |

## 2.3 External Systems and Interfaces

|  |  |  |  |
| --- | --- | --- | --- |
| System | Description | Interface | Phase |
| CEA | Provides cConsol Id mapping, client product enablement, facing entity mapping, and trader-salesperson coverage data | Read-only DB view | 1 |
| Cumulus | Provides client onboarding data, cConsol relationships, and product taxonomy | REST API | 1 |
| ION | Consumes client mapping, tiering and sales coverage data for trading operations | Kafka – ION Bridge | 1 |
| QTP | Uses client tiering data | REST API | 1 |
| BondPort | Consumes client data for cConsol, tiering and facing entity | REST API | 1 |
| TradeWeb | Provides client product enablement and streaming tier data | To be determined | 2 |
| Bloomberg | Provides client product enablement, streaming tier data, and auto-enablement reports | To be determined | 2 |

# 3. Core Use Cases

This section outlines the primary use cases that CRD is designed to address. These use cases form the foundation of the system’s functionality and guide its architectural decisions.

## 3.1 Use Case 1: Salesperson Viewing Coverage Information

**Scenario**

A salesperson needs to see which clients and client traders they are covering for specific products.

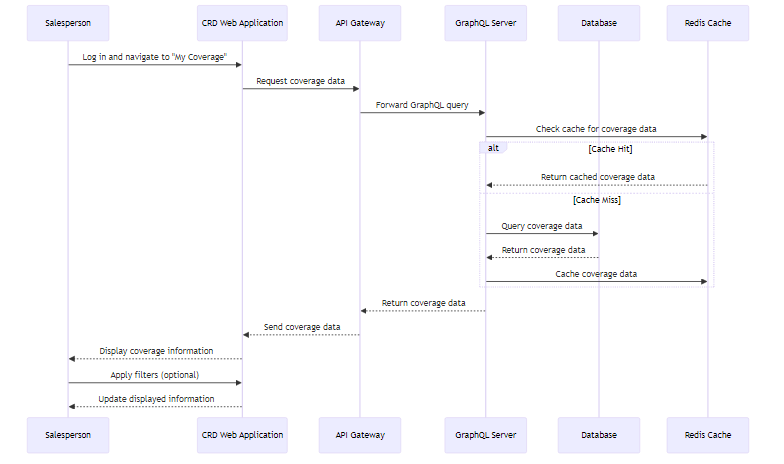
**Actor**

* Salesperson

**Preconditions**

* Salesperson is authenticated in the CRD system
* Coverage data is up to date in the system

**Main Flow**



**UI**

[TODO: Add UI Mockup from Kath]

**Alternative Flows**

* If the salesperson has no assigned coverage, the system displays a message indicating this and provides steps to get coverage assigned.
* If the coverage data fails to load, the system displays an error message and provides an option to retry.

**Postconditions**

* Salesperson has viewed their current coverage assignments.

**Technical Implementation Details**

**Frontend**: React component for displaying coverage information in a tabular format

**Backend**: GraphQL query to fetch salesperson’s coverage data

**Database**: Joins between salespersons, coverage, clients, client traders, and products tables

**Caching**: Redis cache for frequently accessed coverage data to improve response times

**Security**: Ensure the salesperson can only view their own coverage information, using Emidas (Neo) authentication.

## 3.2 Use Case 2: Client Tiering for Incoming Tickets

**Scenario**

The system needs to determine the appropriate tier for an incoming client ticket based on the client, product, and venue.

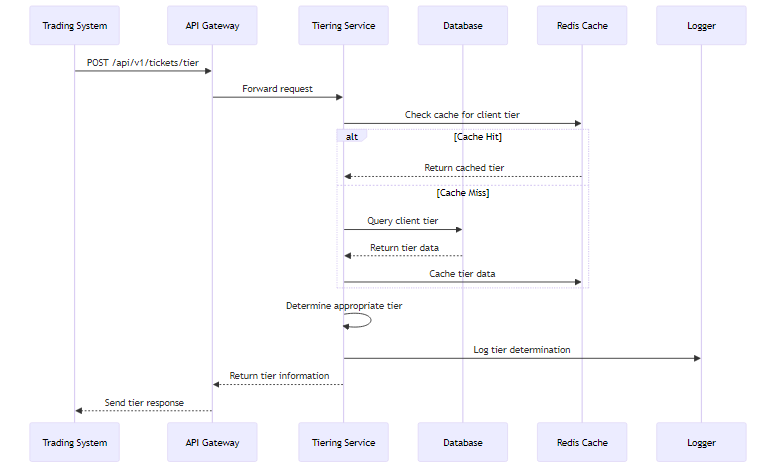
**Actor**

* QTP/ION System

**Preconditions**

* Client tiering data exists and is up to date in the system.
* Incoming ticket Client LEI, Trader, Product, Venue is received from a trading system.

**Main Flow**



**Alternative Flows**

* If Client Not Found
  + System returns Not Found Response message.
  + System logs a warning, and tracks the user and the request details.
* If No Specific Tier Found
  + System uses fallback logic (e.g., client’s general tier for the product type)
  + System logs the use of fallback logic.

**Postconditions**

* Trading system receives the correct tier for the incoming ticket.
* Tier determination is logged for future reference.

**Technical Implementation Details**

**API**: RESTful endpoint for receiving ticket information and returning tier data

**Backend**: Tiering Service to determine appropriate tier based on client, product, and venue

**Database**: Efficient lookup in *tiers* table using composite index

**Caching**: Redis cache for frequently accessed tier data to improve response times

**Performance**: Optimize for low latency, as this may be in the critical path for trade processing.

**Logging**: Structured logging of all tier determinations for audit and analysis

**Security**: SSL Certificate Authentication

## 3.3 Use Case 3: External System (ION) Querying Client Data

**Scenario**

The ION trading system needs to retrieve client data, including cConsol ID, facing entity, and tiering information for a specific LEI and product type.

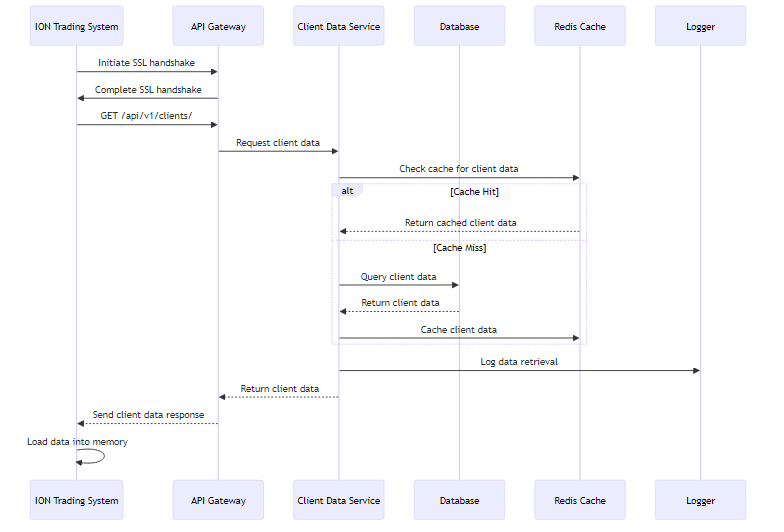
**Actor**

* ION System

**Preconditions**

* ION system has valid SSL Certificate for CRD
* Client data exists in CRD.

**Main Flow**



**Alternative Flows**

* If SSL Certificate authentication fails, the API Gateway rejects the connection ION system logs the authentication failure and alerts the operations team.
* If no client data is found, CRD returns an empty dataset ION system logs the empty dataset and fires the appropriate alerts.

**Postconditions**

* ION system has received the required client data.

**Technical Implementation Details**

**API Endpoint**: GET /api/v1/clients/

**Authentication**: SSL Certificate-based authentication for API requests

**Database** Queries: Joins across clients, client\_products, products, venues, client\_traders, client\_tiers, and facing\_entities tables.

**Response Format**: JSON array structure including lei, product, venue, ctpTraderVenueId, tier, and facingEntity for all clients.

**Performance**: Implement Redis caching for the entire dataset with a 15-minute expiration

**Logging**: Log all external system queries for audit purposes, including requester ID and record count

## 3.4 Use Case 4: Real-time Update of Client Data from CEA

**Scenario**

The CEA system has updated client product enablement information that needs to be reflected in CRD immediately.

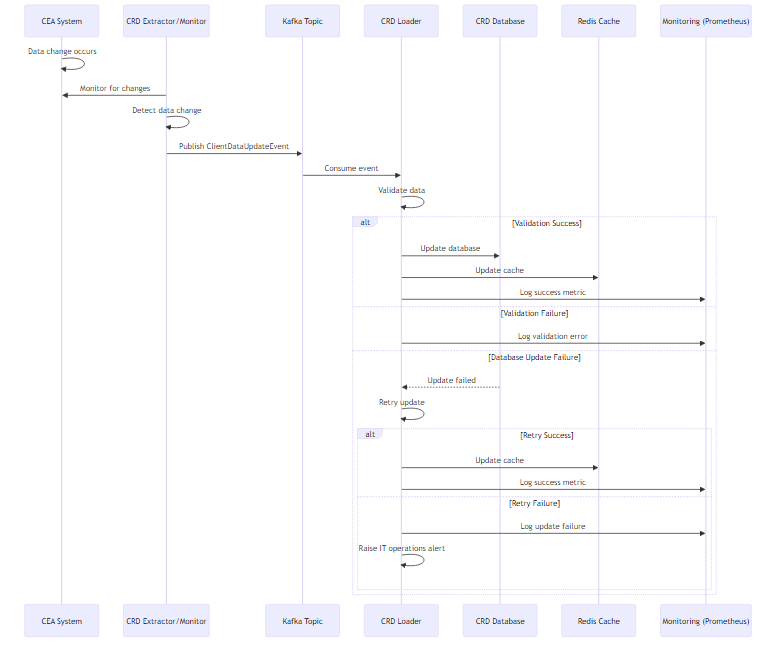
**Actor**

* CEA System
* CRD Extractor/Monitor Service
* CRD Loader

**Preconditions**

* CEA system is operational and accessible by CRD Extractor/Monitor
* Kafka cluster is operational and accessible by CRD components.
* CRD Database is operational and accessible by CRD Loader

**Main Flow**



**Workflow Summary**

1. CEA system updates client data (product enablement, tiering, or coverage)
2. CRD Extractor/Monitor Service detects the change in CEA
3. CRD Extractor/Monitor publishes a "ClientDataUpdateEvent" to a designated Kafka topic
4. CRD Loader consumes the event from Kafka
5. CRD Loader validates the received data
6. CRD Loader updates the CRD database with the new information

**Alternative Flows**

* If the data validation fails, CRD logs an error and flags the record for manual review.
* If the database update fails, CRD attempts a retry and, if unsuccessful, raises an alert for IT operations.

**Postconditions**

CRD database is updated with the latest client information Failed updates or validations are logged and flagged for review.

**Technical Implementation Details**

**Kafka Consumer**: Subscribes to “internal.client.emea.event.<<dataset>>.update” topic.

**Data Validation**: JSON schema validation for incoming event data

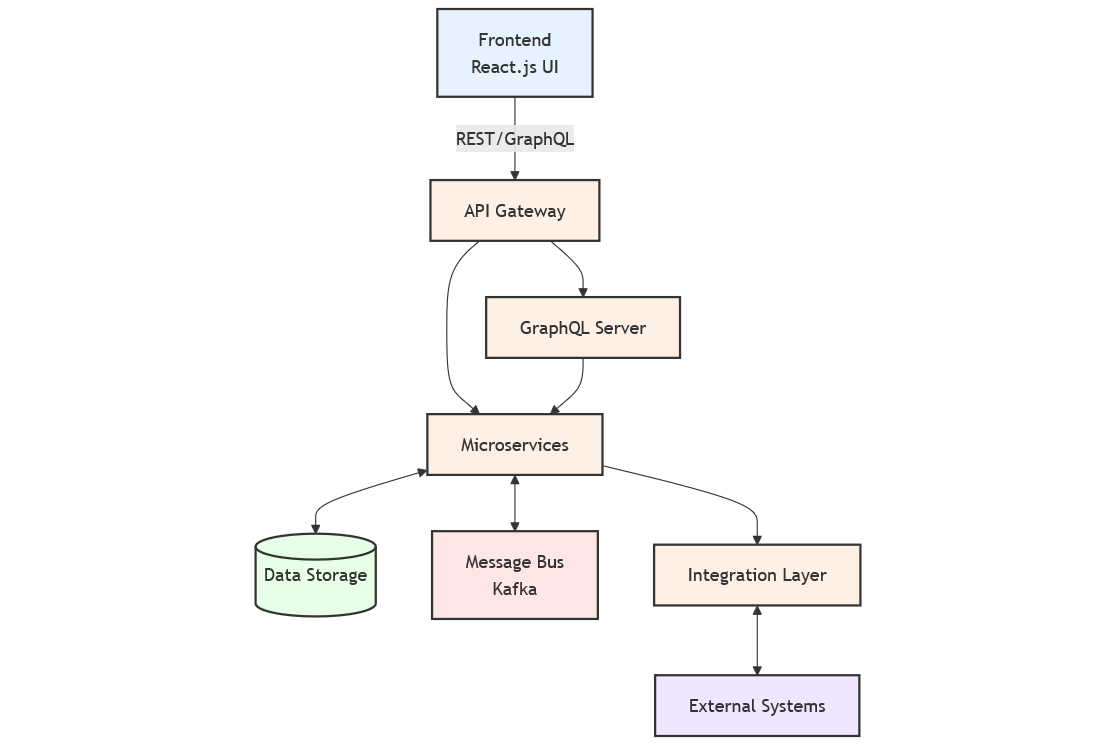
**Database Update**: Upsert operation on any Client, Tiering, Product or Sales Coverage tables.

**Event Publishing**: Internal Kafka producer sends “DataUpdateEvent” to “internal.client.emea.event.<<dataset>>.update” topic

**Error Handling**: Dead Letter Queue for failed event processing

**Monitoring**: Prometheus metrics for tracking event processing latency and success rates

# 4. High-Level Architecture



## 4.1 Key Components

* Frontend (React.js UI)
  + Provides user interface for traders, salespeople and eCST Team.
  + Communicates with backend services via API Gateway
* API Gateway
  + Handles authentication and request routing.
  + Provides a single-entry point for all client-side requests.
* Microservices
  + Client Data Service: Manages core client data (mappings, enablement, and coverage)
  + Tiering Service: Handles client tiering logic and data.
  + Integration Service: Manages communication with external systems.
* Data Storage
  + Microsoft SQL Server: Primary data store for CRD data
  + Redis: Caching layer for frequently accessed data
* Message Bus (Kafka)
  + Facilitates event-driven communication between services.
  + Enables real-time data updates and integration with external systems.
* GraphQL Server
  + Provides a flexible query interface for complex data requests.
  + Integrates with existing microservices to aggregate data efficiently.
* Integration Layer
  + Handles data synchronization with CEA and other external systems.
  + Implements FpML message processing for standardized data exchange.

## 4.2 Technology Stack

* Frontend: TypeScript, React.js, Node.js
* Backend Services: Java, Spring Boot
* API: REST, GraphQL (for complex queries)
* GraphQL: graphql-java, graphql-java-tools
* Database: Microsoft SQL Server
* Cache: Redis
* Message Bus: Apache Kafka
* Containerization: Docker
* Orchestration: Kubernetes
* CI/CD: GitLab CI
* Monitoring: Prometheus, Grafana

## 4.3 Goals and Constraints

### 4.3.1 Security and Compliance Requirements

* UI will be handled by Emidas (NEO), and services will be secured using SSL Certificate Authentication
* Encryption of data in transit and at rest
* Audit logging for all data modifications
* Compliance with bank’s internal security policies and relevant financial regulations (e.g., GDPR, MiFID II)

### 4.3.2 Availability and Reliability

* 99.99% uptime during trading hours
* Fault tolerance and automatic failover for critical components
* Disaster recovery with Recovery Point Objective (RPO) < 5 minutes and Recovery Time Objective (RTO) < 30 minutes

### 4.3.3 Maintainability and Extensibility

* Modular architecture to allow for easy addition of new features
* Comprehensive API documentation
* Consistent coding standards and practices across all components

### 4.3.4 Integration Requirements

* Real-time data synchronization with CEA
* Ability to consume and produce FpML messages
* Support for both synchronous (REST) and asynchronous (Kafka) communication patterns

# 5. Detailed Architecture

## 5.1 Frontend Architecture

The frontend will be built using React.js with TypeScript on the NEO platform, following a component-based architecture. Key aspects include:

* Component Structure
  + Atomic design principles for reusable UI components
  + Container components for managing data and state.
  + Presentation components for rendering UI
* State Management
  + Redux for global state management
  + React Query for server state management and caching.
* API Integration
  + Axios for HTTP requests
  + Custom hooks for encapsulating API logic
* Routing
  + React Router for client-side routing.
* Testing
  + Jest and React Testing Library for unit and integration tests.

Example component structure:

// ClientSearchComponent.tsx  
import React, { useState } from 'react';  
import { useClientSearch } from '../hooks/useClientSearch';  
import SearchInput from './SearchInput';  
import ClientList from './ClientList';  
  
const ClientSearchComponent: React.FC = () => {  
 const [searchTerm, setSearchTerm] = useState('');  
 const { clients, isLoading, error } = useClientSearch(searchTerm);  
  
 return (  
 <div>  
 <SearchInput value={searchTerm} onChange={setSearchTerm} />  
 {isLoading ? (  
 <p>Loading...</p>  
 ) : error ? (  
 <p>Error: {error.message}</p>  
 ) : (  
 <ClientList clients={clients} />  
 )}  
 </div>  
 );  
};  
  
export default ClientSearchComponent;

## 5.2 Backend Architecture

### 5.2.1 General API Information

* Base URL: https://api.crd.neo.ubs-test.net/v1
* Authentication: SSL Certificate Authentication & Emidas
* Entitlements Management:
  + Arcot Id (Sensitive data)
  + LWS - Employee Directory
* Rate Limiting: 1000 requests per minute per API key.
* All responses are in JSON format.
* Use HTTPS for all requests.

### 5.2.2 Authentication

All UI authentications will be handled by NEO (Emidas). Serivce to service authentication will be handled using SSL Certificate Authentication. [NEO Authentication URL]

### 5.2.3 Error Handling

All errors follow this format:

{  
 "error": {  
 "code": "ERROR\_CODE",  
 "message": "A human-readable error message",  
 "details": {} // Optional object with additional error details  
 }  
}

Common HTTP status codes:

* 200: Success
* 400: Bad Request
* 401: Unauthorized
* 403: Forbidden
* 404: Not Found
* 429: Too Many Requests
* 500: Internal Server Error

### 5.2.4 Endpoints

#### 5.2.4.1 Get Salesperson Coverage

Retrieves the coverage information for a salesperson, including both client-level and trader-level coverage.

* **URL**: /coverage
* **Method**: GET
* **URL Params**:
  + salespersonId=[string] (required)
  + productType=[string] (optional)
* **Success Response**:
  + Code: 200
  + Content:
  + {  
     "salesperson": {  
     "id": "SP123",  
     "name": "John Doe"  
     },  
     "coverage": [  
     {  
     "client": {  
     "lei": "LEI123456789",  
     "name": "Acme Corp",  
     "coverageType": "CLIENT",  
     "products": ["FX", "Equities"]  
     }  
     },  
     {  
     "client": {  
     "lei": "LEI987654321",  
     "name": "Global Trading Inc",  
     "traders": [  
     {  
     "id": "TR789",  
     "name": "Alice Trader",  
     "coverageType": "TRADER",  
     "products": ["FX", "Rates"]  
     },  
     {  
     "id": "TR555",  
     "name": "Jack Trader",  
     "coverageType": "TRADER",  
     "products": ["FX"]  
     }]  
     }  
     }  
     ]  
    }
* **Error Response**:
  + Code: 404
  + Content: { "error": { "code": "SALESPERSON\_NOT\_FOUND", "message": "Salesperson not found" } }

#### 5.2.4.2 Get Client Tier for Ticket

Determines the appropriate tier for an incoming client ticket.

* **URL**: /tickets/tier
* **Method**: POST
* **Data Params**:
* {  
   "requestId": "REQ0002",  
   "clientLei": "LEI123456789",  
   "productType": "FX",  
   "venue": "Bloomberg",  
   "traderId": "TR123"  
  }
* **Success Response**:
  + Code: 200
  + Content:
  + {  
     "requestId": "REQ0002",  
     "tier": "Tier1"  
    }
* **Error Response**:
  + Code: 400
  + Content: { "error": { "code": "INVALID\_PRODUCT", "message": "Invalid product type provided" } }

#### 5.2.4.3 Get Client Data

Retrieves comprehensive client data including cConsol ID, facing entity, and tiering information.

* **URL**: /clients/:lei
* **Method**: GET
* **URL Params**:
  + lei=[string] (required)
* **Query Params**:
  + productType=[string] (optional)
* **Success Response**:
  + Code: 200
  + Content:
  + {  
     "ids": [{  
     "domain": "LEI",  
     "value": "LEI123456789",  
     },{  
     "domain": "CE",  
     "value": " CC987654",  
     }],  
     "name": "Acme Corp",  
     "facingEntity": {  
     "id":{  
     "domain": "CE",  
     "value": "CC00122333",  
     },  
     "name": "UBS AG LN"  
     },  
     "tiers": [  
     {  
     "productType": "FX",  
     "venue": "Bloomberg",  
     "tier": "Tier1"  
     }  
     ]  
    }
* **Error Response**:
  + Code: 404
  + Content: { "error": { "code": "CLIENT\_NOT\_FOUND", "message": "Client not found" } }

#### 5.2.4.4 Update Client Tier

Updates the tier for a specific client, product, and venue combination.

* **URL**: /clients/:lei/tiers
* **Method**: PUT
* **URL Params**:
  + lei=[string] (required)
* **Data Params**:
* {  
   "productType": "Equities",  
   "venue": "TradeWeb",  
   "tier": "Tier2"  
  }
* **Success Response**:
  + Code: 200
  + Content:
  + {  
     "message": "Tier updated successfully",  
     "tier": {  
     "clientLei": "LEI123456789",  
     "productType": "Equities",  
     "venue": "TradeWeb",  
     "tier": "Tier2"  
     }  
    }
* **Error Response**:
  + Code: 403
  + Content: { "error": { "code": "INSUFFICIENT\_PERMISSIONS", "message": "User does not have permission to update tiers" } }

#### 5.2.4.5 Get Secondary Coverage

Retrieves secondary coverage information for a specific client trader.

* **URL**: /coverage/secondary
* **Method**: GET
* **Query Params**:
  + clientLei=[string] (required)
  + traderId=[string] (required)
* **Success Response**:
  + Code: 200
  + Content:
  + {  
     "clientTrader": {  
     "id": "TR789",  
     "name": "Alice Trader"  
     },  
     "coverage": [  
     {  
     "salesperson": [{  
     "id": "SP456",  
     "name": "Bob SecondaryRP",  
     "type": "secondary"  
     }],  
     "products": ["FX", "Rates"]  
     }  
     ]  
    }
* **Error Response**:
  + Code: 404
  + Content: { "error": { "code": "TRADER\_NOT\_FOUND", "message": "Client trader not found" } }

#### 5.2.4.6 Pagination

For endpoints that return lists, use cursor-based pagination:

* Request:
  + Add cursor and limit query parameters.
  + Example: GET /clients?cursor=CURSOR123&limit=50
* Response:
  + Include nextCursor in the response for the client to use in subsequent requests.
* Example:
  + {  
     "clients": [...],  
     "nextCursor": "CURSOR123"  
    }

#### 5.2.4.7 Versioning

* The API version is included in the URL (e.g., /v1/)
* When breaking changes are needed, the version number is incremented (e.g., /v2/)

#### 5.2.4.8 Caching

* ETags will be used for caching to reduce unnecessary data transfer.
* Server will include the ETag header in responses.
* Clients can send the If-None-Match header with the ETag value to check if the resource has changed.

#### 5.2.4.9 Service Layer

The service layer will contain the core business logic of the application. Key aspects include:

* Service Boundaries
  + Clear separation of concerns between services
  + Domain-driven design principles for service organization
* Transaction Management
  + Declarative transaction management using Spring’s @Transactional annotation.
* Caching
  + Method-level caching using Spring Cache abstraction backed by Redis.
* Event Publishing
  + Domain events for triggering asynchronous processes.

Example service implementation:

@Service  
public class ClientServiceImpl implements ClientService {  
  
 private final ClientRepository clientRepository;  
 private final KafkaTemplate<String, ClientEvent> kafkaTemplate;  
  
 @Autowired  
 public ClientServiceImpl(ClientRepository clientRepository, KafkaTemplate<String, ClientEvent> kafkaTemplate) {  
 this.clientRepository = clientRepository;  
 this.kafkaTemplate = kafkaTemplate;  
 }  
  
 @Override  
 @Transactional  
 @Cacheable(value = "clients", key = "#lei")  
 public ClientDTO getClientByLEI(String lei) {  
 Client client = clientRepository.findByLEI(lei)  
 .orElseThrow(() -> new ClientNotFoundException("Client not found for LEI: " + lei));  
 return mapToDTO(client);  
 }  
  
 @Override  
 @Transactional  
 @CacheEvict(value = "clients", key = "#clientCreateDTO.lei")  
 public ClientDTO createClient(ClientCreateDTO clientCreateDTO) {  
 Client client = mapToEntity(clientCreateDTO);  
 Client savedClient = clientRepository.save(client);  
 kafkaTemplate.send("client-updates", new ClientCreatedEvent(savedClient.getLEI()));  
 return mapToDTO(savedClient);  
 }  
  
 // Other methods...  
  
 private ClientDTO mapToDTO(Client client) {  
 // Mapping logic  
 }  
  
 private Client mapToEntity(ClientCreateDTO dto) {  
 // Mapping logic  
 }  
}

### 5.2.5 Data Access Layer

The data access layer will use Spring Data JPA for interacting with the Microsoft SQL Server database. Key aspects include:

* Repository Pattern
  + Spring Data JPA repositories for standard CRUD operations.
  + Custom repository methods for complex queries
* Query Optimization
  + Use of database indexes for frequently queried fields
  + Query optimization through JPA named queries and native queries for complex operations.
* Caching Strategy
  + Entity-level caching using JPA second level cache
  + Query result caching for frequently executed queries.

Example repository implementation:

@Repository  
public interface ClientRepository extends JpaRepository<Client, Long> {  
  
 Optional<Client> findByLEI(String lei);  
  
 @Query("SELECT c FROM Client c WHERE c.cConsolId = :cConsolId AND c.productType = :productType")  
 List<Client> findByCConsolIdAndProductType(@Param("cConsolId") String cConsolId, @Param("productType") String productType);  
  
 @Query(value = "SELECT c.\* FROM clients c JOIN client\_tiers ct ON c.id = ct.client\_id WHERE ct.tier = :tier AND ct.product\_type = :productType", nativeQuery = true)  
 List<Client> findClientsByTierAndProductType(@Param("tier") String tier, @Param("productType") String productType);  
}

### 5.2.6 GraphQL Layer

The GraphQL layer sits alongside our REST API, providing an efficient interface for complex, nested queries. Key aspects include:

* Schema Definition: Defined using GraphQL SDL, representing all queryable entities and their relationships.
* Resolvers: Java classes that map GraphQL operations to our domain logic and data fetching.
* DataFetchers: Custom implementations for optimized data retrieval in complex scenarios.
* Integration: Utilizes the same service layer as the REST API, ensuring consistency.

## 5.3 Database Architecture

The database will be implemented using Microsoft SQL Server. Key aspects include:

* Schema Design
  + Normalized structure for efficient data storage and retrieval
  + Use of appropriate data types and constraints
* Indexing Strategy
  + Clustered index on primary keys
  + Non-clustered indexes on frequently queried columns (e.g., LEI, cConsolId)
* Partitioning
  + Table partitioning for large tables:
    - Tiers
    - AlternateId
* Performance Optimization
  + Regular statistics updates
  + Query plan analysis and optimization.

## 5.4 Integration Architecture

### 5.4.1 Upstream Integration

Integration with CEA and Cumulus will be implemented using a combination of database views and event-driven architecture.

* CEA Integration
  + Read-only database view for accessing CEA data.
  + Scheduled job to sync data from CEA to CRD database.
  + Kafka topics for real-time updates
* Cumulus Integration
  + API-based integration for client onboarding data
  + Kafka topics for real-time updates on client and product taxonomy changes

Example integration service:

@Service  
public class CEAIntegrationService {  
  
 private final JdbcTemplate jdbcTemplate;  
 private final ClientRepository clientRepository;  
 private final KafkaTemplate<String, CEAUpdateEvent> kafkaTemplate;  
  
 @Autowired  
 public CEAIntegrationService(JdbcTemplate jdbcTemplate, ClientRepository clientRepository, KafkaTemplate<String, CEAUpdateEvent> kafkaTemplate) {  
 this.jdbcTemplate = jdbcTemplate;  
 this.clientRepository = clientRepository;  
 this.kafkaTemplate = kafkaTemplate;  
 }  
  
 @Scheduled(fixedRate = 3600000) // Run every hour  
 public void syncCEAData() {  
 String sql = "SELECT \* FROM CEA\_CLIENT\_VIEW WHERE last\_updated > :lastSyncTime";  
 List<Map<String, Object>> ceaData = jdbcTemplate.queryForList(sql, Collections.singletonMap("lastSyncTime", getLastSyncTime()));  
  
 for (Map<String, Object> row : ceaData) {  
 Client client = mapCEADataToClient(row);  
 clientRepository.save(client);  
 kafkaTemplate.send("cea-updates", new CEAUpdateEvent(client.getLEI()));  
 }  
  
 updateLastSyncTime();  
 }  
  
 // Helper methods...  
}

### 5.4.2 Downstream Integration

Integration with downstream systems (ION, QTP, BondPort) will be implemented using RESTful APIs and Kafka topics.

* RESTful API
  + Endpoints for querying client data, tiering information, and product enablement
  + Versioned API to support backward compatibility.
* Kafka Topics
  + Real-time event streams for client updates, tier changes, and product enablement changes

Example Kafka producer:

@Component  
public class ClientUpdateProducer {  
  
 private final KafkaTemplate<String, ClientUpdateEvent> kafkaTemplate;  
  
 @Autowired  
 public ClientUpdateProducer(KafkaTemplate<String, ClientUpdateEvent> kafkaTemplate) {  
 this.kafkaTemplate = kafkaTemplate;  
 }  
  
 public void sendClientUpdate(ClientUpdateEvent event) {  
 kafkaTemplate.send("client-updates", event.getClientLEI(), event);  
 }  
}

## 5.5 Event-Driven Architecture

The CRD system will use Apache Kafka as the backbone for event-driven architecture.

* Kafka Topic Naming Convention:
  + *<classification>.<domain>.<region>.<type>.<dataset>.<data>*
* Kafka Topic Design (Naming Convention)
  + internal.client.emea.event.client.update: For general client data changes
  + internal.client.emea.event.tiering.update: For changes in client tiering
  + internal.client.emea.event.product.update: For changes in product enablement
* Event Schema Management
  + Use of Avro for schema definition and evolution
  + Implementation of Schema Registry for version control
* Kafka Streams
  + Real-time data processing for analytics and derived data

Example Kafka consumer:

@Component  
public class ClientUpdateConsumer {  
  
 private final ClientService clientService;  
  
 @Autowired  
 public ClientUpdateConsumer(ClientService clientService) {  
 this.clientService = clientService;  
 }  
  
 @KafkaListener(topics = "internal.client.emea.event.client.update", groupId = "crd-consumer-group")  
 public void consumeClientUpdate(ClientUpdateEvent event) {  
 clientService.handleClientUpdate(event);  
 }  
}

# 6. Data Model

## 6.1 Entity Relationship Overview

The CRD system employs a complex, interconnected data model designed to handle the intricacies of client relationships, product offerings, and trading operations. The model is characterized by its flexibility, hierarchical structures, and ability to support multi-jurisdictional operations.

The following diagrams illustrate the comprehensive structure of the CRD data model:

TODO:: Complete CRD Data Model

TODO:: Tiering

TODO:: Alternate Ids

## 6.2 Key Entities and Relationships

* Client
  + Represents the core client entity with attributes: id, name, tier\_id, parent\_client\_id, jurisdiction\_id.
  + Supports hierarchical client structures through self-referential parent\_client\_id.
  + Associated with Jurisdiction and Tier entities.
* Person
  + Stores individual contact information: id, first\_name, last\_name, type.
  + Linked to clients through associative entities.
* ProductItem
  + Represents financial products: id, name, description, parent\_id, type\_id, path.
  + Supports hierarchical product structures.
  + Connected to venues through VenueProduct entity.
* Domain and AlternateId
  + Domain: id, name, group, is\_internal, description
  + AlternateId: Flexible identification system linking various entities across domains.
  + Crucial for integration with external systems and supporting multiple identification schemes.
* UBSEntity
  + Represents internal organizational units: id, entity\_name, jurisdiction\_id.
  + Linked to clients through ClientAccount entity.
* DeskClient
  + Represents a client in the context of a specific desk: id, desk\_id, client\_id, tier\_id.
  + Central to the tiering system and client-desk relationships
* Venue and VenueProduct
  + Venue: Represents trading venues or platforms
  + VenueProduct: Links products to specific venues, allowing for venue-specific product configurations.
* Employee and Coverage
  + Employee: Stores information about the firms’ employees
  + DeskEmployee: Links employees to specific desks
  + SalesCoverage: Manages the relationship between employees, clients, and products for sales activities.
* Tier
  + Central to the client tiering system: id, name, description, rank
  + Associated with multiple entities including Client, DeskClient, and DeskClientProduct

## 6.3 Key Relationships

* *Client* to *DeskClient*: One-to-Many, representing a client’s relationship with different desks.
* *Client* to *ClientVenueAccount*: One-to-Many, linking clients to their accounts on various venues.
* *ProductItem* to *VenueProduct*: One-to-Many, associating products with venues
* *DeskClient* to *DeskClientProduct*: One-to-Many, defining product-specific attributes for clients on each desk.
* *Employee* to *SalesCoverage*: One-to-Many, representing an employee’s coverage of various clients and products.
* Tier to multiple entities: One-to-Many, implemented in *Client*, *DeskClient*, and *DeskClientProduct* for flexible tiering.

## 6.4 Data Management Strategies

* Identification and Cross-referencing
  + Utilize the AlternateId entity to manage multiple identification systems across different domains.
  + Implement efficient indexing on the AlternateId table to ensure fast cross-referencing.
* Hierarchical Data Handling
  + Implement materialized path or nested set model for efficient querying of hierarchical client and product structures
  + Use recursive queries for traversing hierarchical data when necessary
* Tiering System
  + Implement tiering at multiple levels (Client, DeskClient, DeskClientProduct) for maximum flexibility
  + Use database views to simplify complex tiering queries
* Jurisdictional Data Management
  + Enforce jurisdictional data segregation through database-level access controls
  + Implement a caching strategy that respects jurisdictional boundaries
* Historical Data and Auditing
  + Implement temporal data management for key entities to maintain historical records
  + Use trigger-based audit logging for tracking changes to critical data

## 6.5 Performance Optimization

* Indexing Strategy
  + Create composite indexes on frequently joined columns
  + Implement partial indexes for large tables where only a subset of rows are regularly queried
* Partitioning
  + Partition large tables (e.g., AlternateId, TicketRoutingRules) based on appropriate criteria such as domain or date ranges
* Materialized Views
  + Create materialized views for complex, frequently-accessed data combinations
  + Implement a refresh strategy that balances data freshness with system performance
* Query Optimization
  + Develop and maintain a set of optimized stored procedures for complex queries
  + Use query hints where necessary to guide the query optimizer

# 7. API Documentation

## 7.1 RESTful API Endpoints

### 7.1.1 Client Management

* GET /api/v1/clients/{lei}
  + Retrieve client details by LEI
  + Parameters:
    - lei (path): Legal Entity Identifier
  + Response: ClientDTO
* POST /api/v1/clients
  + Create a new client
  + Request Body: ClientCreateDTO
  + Response: ClientDTO
* PUT /api/v1/clients/{lei}
  + Update client details
  + Parameters:
    - lei (path): Legal Entity Identifier
  + Request Body: ClientUpdateDTO
  + Response: ClientDTO
* GET /api/v1/clients/search
  + Search for clients
  + Parameters:
    - query (query): Search term
    - page (query): Page number
    - size (query): Page size
  + Response: Page

### 7.1.2 Tiering Management

* GET /api/v1/tiers/{lei}
  + Retrieve tiering information for a client
  + Parameters:
    - lei (path): Legal Entity Identifier
  + Response: List
* POST /api/v1/tiers
  + Create a new tier for a client
  + Request Body: TierCreateDTO
  + Response: TierDTO
* PUT /api/v1/tiers/{tierId}
  + Update tier information
  + Parameters:
    - tierId (path): Tier identifier
  + Request Body: TierUpdateDTO
  + Response: TierDTO

### 7.1.3 Product Enablement

* GET /api/v1/product-enablement/{lei}
  + Retrieve product enablement for a client
  + Parameters:
    - lei (path): Legal Entity Identifier
  + Response: List
* POST /api/v1/product-enablement
  + Enable a product for a client
  + Request Body: ProductEnablementCreateDTO
  + Response: ProductEnablementDTO
* DELETE /api/v1/product-enablement/{enablementId}
  + Disable a product for a client
  + Parameters:
    - enablementId (path): Product enablement identifier
  + Response: 204 No Content

### 7.1.4 Sales Coverage Management

* GET /api/v1/coverage/{lei}
  + Retrieve sales coverage information for a client
  + Parameters:
    - lei (path): Legal Entity Identifier
  + Query Parameters:
    - productType (query, optional): Filter by product type
  + Response: List<CoverageDTO>
* POST /api/v1/coverage
  + Assign sales coverage for a client
  + Request Body: CoverageCreateDTO
  + Response: CoverageDTO
* PUT /api/v1/coverage/{coverageId}
  + Update sales coverage information
  + Parameters:
    - coverageId (path): Coverage identifier
  + Request Body: CoverageUpdateDTO
  + Response: CoverageDTO
* DELETE /api/v1/coverage/{coverageId}
  + Remove sales coverage assignment
  + Parameters:
    - coverageId (path): Coverage identifier
  + Response: 204 No Content
* GET /api/v1/coverage/salesperson/{salespersonId}
  + Retrieve all coverage assignments for a salesperson
  + Parameters:
    - salespersonId (path): Salesperson identifier
  + Query Parameters:
    - productType (query, optional): Filter by product type
    - page (query, optional): Page number
    - size (query, optional): Page size
  + Response: Page<CoverageDTO>
* GET /api/v1/coverage/trader/{traderId}
  + Retrieve sales coverage for a specific trader
  + Parameters:
    - traderId (path): Trader identifier
  + Query Parameters:
    - productType (query, optional): Filter by product type
  + Response: List<CoverageDTO>

## 7.2 GraphQL API

### 7.2.1 Rationale for GraphQL Alongside REST

While our primary API will be RESTful, we are incorporating GraphQL to address specific challenges and use cases within the CRD system:

* Complex Data Relationships: The CRD system deals with intricate relationships between clients, tiers, product enablement, traders, and salespersons. GraphQL’s ability to request nested, related data in a single query can significantly reduce the number of API calls needed for complex operations.
* Flexibility for Frontend Development: GraphQL allows the frontend to request exactly the data it needs, reducing over-fetching and under-fetching of data. This is particularly beneficial for the React-based UI, enabling more efficient data loading and management.
* Evolving Data Requirements: As the CRD system grows and evolves, GraphQL’s schema can be extended without versioning the entire API, providing more flexibility for future enhancements.
* Aggregation of Multiple Data Sources: GraphQL can act as a data aggregation layer, combining data from various microservices (Client Data Service, Tiering Service, etc.) into a single, coherent API for frontend consumption.

### 7.2.2 GraphQL Schema Definition

Below is a more detailed GraphQL schema definition for the CRD system:

type Query {  
 client(lei: String!): Client  
 searchClients(query: String!, page: Int, size: Int): ClientConnection  
 clientTiers(lei: String!, productType: String): [Tier]  
 productEnablement(lei: String!, venue: String): [ProductEnablement]  
}  
  
type Mutation {  
 updateClientTier(input: UpdateTierInput!): Tier  
 enableProduct(input: EnableProductInput!): ProductEnablement  
}  
  
type Client {  
 lei: String!  
 cConsolId: String!  
 name: String!  
 tiers: [Tier]  
 productEnablements: [ProductEnablement]  
 traders: [ClientTrader]  
 facingEntities: [FacingEntity]  
}  
  
type Tier {  
 id: ID!  
 clientLei: String!  
 productType: String!  
 venue: String!  
 tier: String!  
 createdAt: DateTime!  
 updatedAt: DateTime!  
}  
  
type ProductEnablement {  
 id: ID!  
 clientLei: String!  
 productType: String!  
 venue: String!  
 enabled: Boolean!  
 createdAt: DateTime!  
 updatedAt: DateTime!  
}  
  
type ClientTrader {  
 id: ID!  
 name: String!  
 coverage: [Coverage]  
}  
  
type Coverage {  
 productType: String!  
 salesperson: Salesperson!  
}  
  
type Salesperson {  
 id: ID!  
 name: String!  
 webSSOId: String!  
}  
  
type FacingEntity {  
 id: ID!  
 name: String!  
 productType: String!  
}  
  
type ClientConnection {  
 edges: [ClientEdge]  
 pageInfo: PageInfo!  
}  
  
type ClientEdge {  
 node: Client  
 cursor: String!  
}  
  
type PageInfo {  
 hasNextPage: Boolean!  
 endCursor: String!  
}  
  
input UpdateTierInput {  
 clientLei: String!  
 productType: String!  
 venue: String!  
 tier: String!  
}  
  
input EnableProductInput {  
 clientLei: String!  
 productType: String!  
 venue: String!  
 enabled: Boolean!  
}  
  
scalar DateTime

### 7.2.3 Specific Use Cases for GraphQL Queries

Comprehensive Client View: Retrieve all relevant information about a client in a single query, including tiers, product enablements, and associated traders.

* query GetClientDetails($lei: String!) {  
   client(lei: $lei) {  
   name  
   cConsolId  
   tiers {  
   productType  
   venue  
   tier  
   }  
   productEnablements {  
   productType  
   venue  
   enabled  
   }  
   traders {  
   name  
   coverage {  
   productType  
   salesperson {  
   name  
   webSSOId  
   }  
   }  
   }  
   facingEntities {  
   name  
   productType  
   }  
   }  
  }
* Filtered Tiering Information: Fetch tiering data for a specific client and product type across all venues.
* query GetClientTiers($lei: String!, $productType: String!) {  
   clientTiers(lei: $lei, productType: $productType) {  
   venue  
   tier  
   }  
  }
* Client Search with Pagination: Search for clients with paginated results.
* query SearchClients($query: String!, $page: Int, $size: Int) {  
   searchClients(query: $query, page: $page, size: $size) {  
   edges {  
   node {  
   lei  
   name  
   cConsolId  
   }  
   cursor  
   }  
   pageInfo {  
   hasNextPage  
   endCursor  
   }  
   }  
  }

### 7.2.4 GraphQL Performance and Security Considerations

Query Complexity and Depth Limiting: Implement query complexity analysis and depth limiting to prevent resource-intensive queries. We’ll use the graphql-java-extended-validation library to enforce these limits.

Dataloaders for N+1 Query Problem: Utilize DataLoader pattern (implemented with java-dataloader) to batch and cache database queries, mitigating the N+1 query problem common in GraphQL implementations.

Caching Strategy: Implement field-level caching using graphql-java-extended-scalars to cache frequently accessed, relatively static data (e.g., client basic information).

Authorization: Integrate with the existing Spring Security setup to ensure proper authorization for GraphQL queries and mutations. Implement field-level security where necessary.

Rate Limiting: Apply rate limiting at the GraphQL API level to prevent abuse. This will be implemented using a custom directive and the bucket4j library.

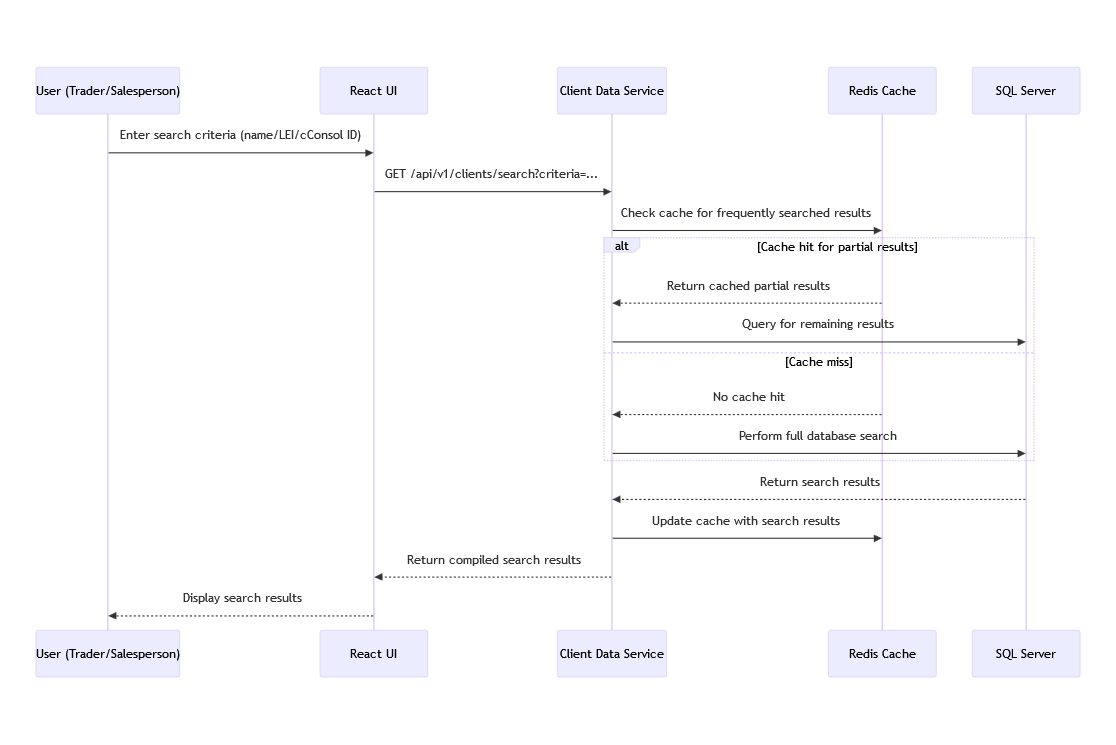
# 8. Data Flow and Sequence Diagrams

## 8.1 Client Search Flow

**Query Scenario**

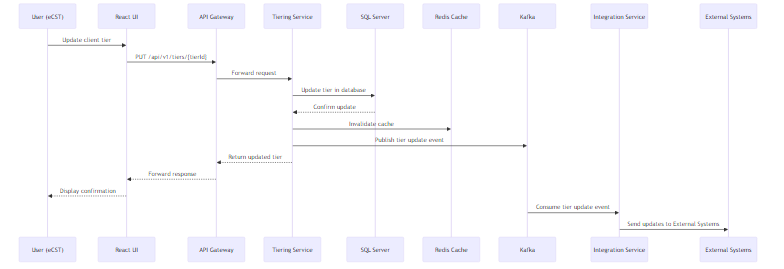
This scenario demonstrates a complex client search process where a user (e.g., a trader or salesperson) can search for a client using multiple identifiers such as name, LEI (Legal Entity Identifier), cConsol ID, or a combination of these.

The search is performed across the CRD system without any direct connection to external systems.



## 8.2 Tiering Update Flow

Query Scenario This scenario demonstrates the process of updating a client’s tier for a specific product type and venue by an eCST (electronic Client Service Team) user, and how this update propagates through the system to external systems.



Key Benefits Illustrated

*Real-time Updates*: The event-driven architecture ensures that tier changes are immediately reflected across the system and propagated to relevant external systems.

*Decoupling*: The use of Kafka for event publishing decouples the Tiering Service from the Integration Service, allowing for independent scaling and maintenance.

*Consistency*: The flow ensures that all parts of the system, including caches and external systems, are updated following a tier change.

*Scalability*: The asynchronous nature of event processing allows the system to handle high volumes of updates without impacting the performance of the user-facing components.

*Audit Trail*: Publishing events to Kafka provides a natural audit trail of all tier changes, which can be consumed for compliance and reporting purposes.

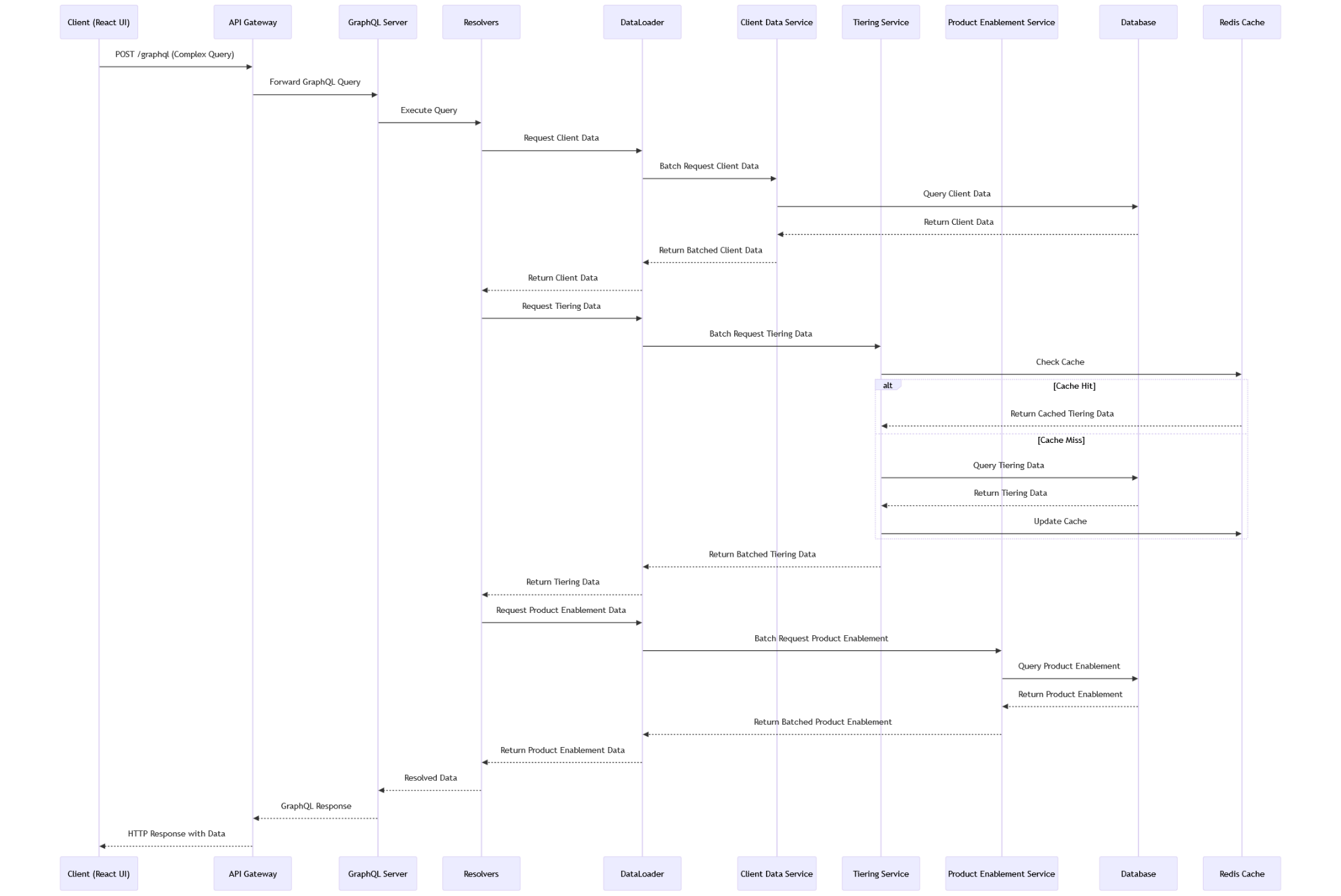
*Cache Management*: The immediate cache invalidation ensures that subsequent queries will fetch the most up-to-date tiering information.

*Streamlined Integration*: The Integration Service directly updates external systems, ensuring efficient propagation of changes without unnecessary database operations.

## 8.3 Complex GraphQL Query Flow

This section describes the flow of a complex GraphQL query through the CRD system, demonstrating how GraphQL efficiently handles nested data requests and utilizes various system components.

*Query Scenario* The example illustrates a complex query that retrieves detailed information about a client, including their tiering data and product enablements across multiple venues.



1. The Client (React UI) sends a POST request to the API Gateway with a complex GraphQL query.
2. The API Gateway forwards the GraphQL query to the GraphQL Server.
3. The GraphQL Server initiates query execution by passing it to the Resolvers.
4. The Resolvers request client data from the DataLoader.
5. The DataLoader batches the client data request and sends it to the Client Data Service.
6. The Client Data Service queries the Database for the requested client data.
7. The Database returns the client data to the Client Data Service.
8. The Client Data Service returns the batched client data to the DataLoader.
9. The DataLoader returns the client data to the Resolvers.
10. The Resolvers then request tiering data from the DataLoader.
11. The DataLoader batches the tiering data request and sends it to the Tiering Service.
12. The Tiering Service checks the Redis Cache for the requested data.
13. If there's a cache hit:
    1. The Redis Cache returns the cached tiering data to the Tiering Service.
    2. If there's a cache miss:
    3. The Tiering Service queries the Database for the tiering data.
    4. The Database returns the tiering data to the Tiering Service.
    5. The Tiering Service updates the Redis Cache with the new data.
14. The Tiering Service returns the batched tiering data to the DataLoader.
15. The DataLoader returns the tiering data to the Resolvers.
16. The Resolvers request product enablement data from the DataLoader.
17. The DataLoader batches the product enablement request and sends it to the Product Enablement Service.
18. The Product Enablement Service queries the Database for the product enablement data.
19. The Database returns the product enablement data to the Product Enablement Service.
20. The Product Enablement Service returns the batched product enablement data to the DataLoader.
21. The DataLoader returns the product enablement data to the Resolvers.
22. The Resolvers compile all the resolved data and return it to the GraphQL Server.
23. The GraphQL Server formats the response and sends it back to the API Gateway.
24. Finally, the API Gateway returns the HTTP response with the requested data to the Client (React UI).

Key Benefits Illustrated

*Efficient Data Fetching*: The use of DataLoader demonstrates how GraphQL can batch multiple requests to backend services, reducing the number of database queries and improving performance.

*Caching*: The flow shows caching in action with the Tiering Service, illustrating how frequently accessed data can be quickly retrieved without hitting the database. Flexible

*Data Aggregation*: The diagram shows how data from multiple services (Client Data, Tiering, Product Enablement) can be combined in a single query, reducing the need for multiple API calls from the client.

*Reduced Over-fetching*: Unlike a REST API where each endpoint might return a fixed data structure, the GraphQL query only retrieves the specific data requested by the client.

# 9. References

#### Mermaid Diagram for 2. System Context

graph LR  
 subgraph Users  
 T[Traders]  
 S[Salespeople]  
 E[eCST]  
 end  
  
 subgraph Upstream Systems  
 CEA[CEA]  
 CM[Cumulus]  
 end  
  
 subgraph Future Integrations  
 TW[TradeWeb]  
 BB[Bloomberg]  
 end  
  
 CRD\_System[CRD System]  
  
 subgraph Downstream Systems  
 ION[ION]  
 QTP[QTP]  
 BP[BondPort]  
 end  
  
 T --> CRD\_System  
 S --> CRD\_System  
 E --> CRD\_System  
  
 CEA --> CRD\_System  
 CM --> CRD\_System  
  
 CRD\_System --> ION  
 CRD\_System --> QTP  
 CRD\_System --> BP  
  
 TW -.-> CRD\_System  
 BB -.-> CRD\_System  
  
 classDef users fill:#e6f3ff,stroke:#333,stroke-width:2px;  
 classDef system fill:#fff0e6,stroke:#333,stroke-width:2px;  
 classDef upstream fill:#e6ffe6,stroke:#333,stroke-width:2px;  
 classDef downstream fill:#ffe6e6,stroke:#333,stroke-width:2px;  
 classDef future fill:#f0e6ff,stroke:#333,stroke-width:2px,stroke-dasharray: 5 5;  
  
 class T,S,E users;  
 class CRD\_System system;  
 class CEA,CM upstream;  
 class ION,QTP,BP downstream;  
 class TW,BB future;

#### Mermaid Diagram for 3.1 Use Case 1: Salespeson Viewing Coverage Information

sequenceDiagram  
 participant S as Salesperson  
 participant UI as CRD Web Application  
 participant API as API Gateway  
 participant GS as GraphQL Server  
 participant DB as Database  
 participant RC as Redis Cache  
   
 S->>UI: Log in and navigate to "My Coverage"  
 UI->>API: Request coverage data  
 API->>GS: Forward GraphQL query  
 GS->>RC: Check cache for coverage data  
 alt Cache Hit  
 RC-->>GS: Return cached coverage data  
 else Cache Miss  
 GS->>DB: Query coverage data  
 DB-->>GS: Return coverage data  
 GS->>RC: Cache coverage data  
 end  
 GS-->>API: Return coverage data  
 API-->>UI: Send coverage data  
 UI-->>S: Display coverage information  
 S->>UI: Apply filters (optional)  
 UI-->>S: Update displayed information

#### Mermaid Diagram for 3.2 Use Case 2: Client Tiering for Incoming Tickets

sequenceDiagram  
 participant TS as Trading System  
 participant API as API Gateway  
 participant TiS as Tiering Service  
 participant DB as Database  
 participant RC as Redis Cache  
 participant L as Logger  
  
 TS->>API: POST /api/v1/tickets/tier  
 API->>TiS: Forward request  
 TiS->>RC: Check cache for client tier  
 alt Cache Hit  
 RC-->>TiS: Return cached tier  
 else Cache Miss  
 TiS->>DB: Query client tier  
 DB-->>TiS: Return tier data  
 TiS->>RC: Cache tier data  
 end  
 TiS->>TiS: Determine appropriate tier  
 TiS->>L: Log tier determination  
 TiS-->>API: Return tier information  
 API-->>TS: Send tier response

#### Mermaid Diagram for 3.3 Use Case 3: External System (ION) Querying Client Data

sequenceDiagram  
 participant ION as ION Trading System  
 participant API as API Gateway  
 participant CDS as Client Data Service  
 participant DB as Database  
 participant RC as Redis Cache  
 participant L as Logger  
  
 ION->>API: Initiate SSL handshake  
 API->>ION: Complete SSL handshake  
 ION->>API: GET /api/v1/clients/  
 API->>CDS: Request client data  
 CDS->>RC: Check cache for client data  
 alt Cache Hit  
 RC-->>CDS: Return cached client data  
 else Cache Miss  
 CDS->>DB: Query client data  
 DB-->>CDS: Return client data  
 CDS->>RC: Cache client data  
 end  
 CDS->>L: Log data retrieval  
 CDS-->>API: Return client data  
 API-->>ION: Send client data response  
 ION->>ION: Load data into memory

#### Mermaid Diagram for 3.4 Use Case 4: Real-time Update of Client Data from CEA

sequenceDiagram  
 participant CEA as CEA System  
 participant EM as CRD Extractor/Monitor  
 participant KT as Kafka Topic  
 participant CL as CRD Loader  
 participant DB as CRD Database  
 participant RC as Redis Cache  
 participant M as Monitoring (Prometheus)  
  
 CEA->>CEA: Data change occurs  
 EM->>CEA: Monitor for changes  
 EM->>EM: Detect data change  
 EM->>KT: Publish ClientDataUpdateEvent  
 KT->>CL: Consume event  
 CL->>CL: Validate data  
 alt Validation Success  
 CL->>DB: Update database  
 CL->>RC: Update cache  
 CL->>M: Log success metric  
 else Validation Failure  
 CL->>M: Log validation error  
 else Database Update Failure  
 DB-->>CL: Update failed  
 CL->>CL: Retry update  
 alt Retry Success  
 CL->>RC: Update cache  
 CL->>M: Log success metric  
 else Retry Failure  
 CL->>M: Log update failure  
 CL->>CL: Raise IT operations alert  
 end  
 end

#### Mermaid Diagram for 4. High-Level Architecture

graph TB

UI[Frontend<br>React.js UI] -->|REST/GraphQL| AG[API Gateway]

AG --> GQL[GraphQL Server]

AG --> MS[Microservices]

GQL --> MS

MS <--> DB[(Data Storage)]

MS <--> MB[Message Bus<br>Kafka]

MS --> IL[Integration Layer]

IL <--> ES[External Systems]

classDef frontend fill:#e6f3ff,stroke:#333,stroke-width:2px;

classDef backend fill:#fff0e6,stroke:#333,stroke-width:2px;

classDef storage fill:#e6ffe6,stroke:#333,stroke-width:2px;

classDef messaging fill:#ffe6e6,stroke:#333,stroke-width:2px;

classDef external fill:#f0e6ff,stroke:#333,stroke-width:2px;

class UI frontend;

class AG,GQL,MS,IL backend;

class DB storage;

class MB messaging;

class ES external;

#### Mermaid diagram for 8.1 Client Search flow

sequenceDiagram

participant U as User (Trader/Salesperson)

participant UI as React UI

participant CDS as Client Data Service

participant RC as Redis Cache

participant DB as SQL Server

U->>UI: Enter search criteria (name/LEI/cConsol ID)

UI->>CDS: GET /api/v1/clients/search?criteria=...

CDS->>RC: Check cache for frequently searched results

alt Cache hit for partial results

RC-->>CDS: Return cached partial results

CDS->>DB: Query for remaining results

else Cache miss

RC-->>CDS: No cache hit

CDS->>DB: Perform full database search

end

DB-->>CDS: Return search results

CDS->>RC: Update cache with search results

CDS-->>UI: Return compiled search results

UI-->>U: Display search results

#### Mermaid Diagram for 8.2 Tiering Update Flow

sequenceDiagram

participant U as User (eCST)

participant UI as React UI

participant AG as API Gateway

participant TS as Tiering Service

participant DB as SQL Server

participant RC as Redis Cache

participant K as Kafka

participant IS as Integration Service

participant ES as External Systems

U->>UI: Update client tier

UI->>AG: PUT /api/v1/tiers/{tierId}

AG->>TS: Forward request

TS->>DB: Update tier in database

DB-->>TS: Confirm update

TS->>RC: Invalidate cache

TS->>K: Publish tier update event

TS-->>AG: Return updated tier

AG-->>UI: Forward response

UI-->>U: Display confirmation

K->>IS: Consume tier update event

IS->>ES: Send updates to External Systems