

API Data Modeling & Schema Design

Reference standard for a fictional retail-banking integration API

Prepared by	Oksana Lisovska
Document type	Technical architecture / business-analysis work sample
Version	1.0
Date	September 17, 2026
Classification	Portfolio Sample / Public

IMPORTANT — PORTFOLIO USE ONLY

This document is an illustrative work sample. It is not an official CIBC document, does not represent CIBC architecture standards, and contains no proprietary CIBC data, system names, credentials, or confidential information.

1. Executive Summary

This reference standard describes a practical approach to API data modeling and schema design for a banking integration. The goal is to create stable, understandable contracts between channels, services, and downstream systems while preserving data quality, backward compatibility, security, and traceability.

The design uses a canonical API model so that consumer applications do not need to understand internal database structures. External contracts are expressed through OpenAPI and JSON Schema; mappings to system-of-record structures are maintained as a separate integration concern.

1.1 Objectives

- Define reusable modeling conventions for identifiers, money, dates, addresses, parties, accounts, and transactions.
- Separate API contracts from internal persistence models to reduce coupling and simplify change.
- Establish validation, versioning, error handling, and compatibility rules that support safe delivery.
- Provide traceable mappings from business requirements to API fields, source data, transformation rules, and tests.

1.2 Scope

The sample focuses on a read-oriented Account & Transaction API. The same principles apply to payment initiation, customer servicing, onboarding, product, and reporting APIs.

2. Modeling Principles

Principle	Application
Business meaning first	Name entities and fields according to business meaning, not source-table names or legacy abbreviations.
Canonical boundary	Expose a stable contract at the API boundary and isolate source-specific transformations behind the service.
Explicit semantics	Document units, code sets, time zones, optionality, null behavior, and field ownership.
Backward compatibility	Additive changes are preferred. Breaking changes require a new major version or a controlled migration.
Minimal disclosure	Return only data required for the consumer use case; sensitive attributes are excluded or masked by default.
Traceable transformations	Every derived field should have a documented source and deterministic transformation rule.

2.1 Conceptual Data Model

The canonical model is intentionally smaller than a core-banking data model. It exposes the business concepts needed by consuming applications while hiding implementation-specific tables and keys.

Entity	Key	Cardinality	Related entity	Rule
Customer	customerid	1 : N	Account	A customer may hold multiple accounts.
Account	accountid	1 : N	Transaction	An account has zero or more posted/pending transactions.
Account	productCode	N : 1	ProductReference	Product metadata is referenced by stable code.
Customer	customerid	1 : N	Address	Addresses are modeled as typed, structured value objects.

2.2 Entity Responsibilities

- Customer: identifies the party in the API context; excludes internal risk or segmentation attributes unless explicitly required.

- *Account*: represents the customer-facing financial account and its status, currency, product reference, and balances.
- *Transaction*: represents posted or pending financial activity using normalized amount, date, type, and description fields.
- *ProductReference*: provides non-sensitive product metadata such as *productCode* and *displayName*.
- *Address*: structured value object with type, lines, city, region, postalCode, and countryCode.

3. Schema Conventions

Element	Representation	Example	Rule
Identifiers	String; opaque to clients	customerId, accountId	Do not expose database sequence values when avoidable.
Dates	YYYY-MM-DD	statementDate	Use date when time-of-day has no business meaning.
Timestamps	RFC 3339 / UTC	createdAt	Example: 2026-09-17T14:30:00Z.
Currency	ISO 4217	CAD, USD	Three-letter uppercase code.
Country	ISO 3166-1 alpha-2	CA, US	Two-letter uppercase code.
Money	Object	amount + currency	Avoid floating-point-only semantics; document scale/precision.
Enums	Stable string codes	ACTIVE, CLOSED	Do not use UI display labels as protocol values.
Booleans	true / false	isJoint	Avoid Y/N and 0/1 in external contracts.

3.1 Required, Optional, and Nullable

Required describes whether a property must be present in a payload. Nullable describes whether the value may explicitly be null. These concepts are independent and must be documented separately.

- Required + non-null: the consumer can rely on a meaningful value being present.
- Optional + non-null: property may be omitted; if present it must contain a valid value.
- Optional + nullable: use only when omission and explicit null have distinct business meaning.
- Avoid returning empty strings to represent missing values unless mandated by a legacy contract.

3.2 Money Value Object

Canonical representation

```
{
  "amount": "1250.42",
  "currency": "CAD"
}
```

Representing amount as a decimal string at the API boundary prevents binary floating-point ambiguity. The schema defines the accepted numeric pattern and the service converts to an appropriate decimal type internally.

3.3 Naming Rules

- Use *lowerCamelCase* for JSON properties and nouns for resources.
- Use descriptive names: *availableBalance* instead of *avlBal*.
- Use *Id* suffix consistently for identifiers: *customerId*, *accountId*, *transactionId*.
- Do not encode data type in field names (for example, *strCustomerName*).

4. Example API Payload

Illustrative endpoint: `GET /v1/customers/{customerId}/accounts/{accountId}`

Example response — fictional data

```
{
  "accountId": "acc_7f3c2a91",
  "customerId": "cus_91bd08af",
  "productCode": "CHQ-PERSONAL",
  "displayName": "Everyday Chequing",
  "status": "ACTIVE",
  "currency": "CAD",
  "balances": {
    "current": { "amount": "2450.18", "currency": "CAD" },
    "available": { "amount": "2210.18", "currency": "CAD" }
  },
  "openedDate": "2022-03-14",
  "links": {
    "transactions": "/v1/accounts/acc_7f3c2a91/transactions"
  }
}
```

4.1 JSON Schema / OpenAPI Excerpt

Account:

```
type: object
required: [accountId, productCode, status, currency, balances]
properties:
  accountId:
    type: string
    minLength: 1
  status:
    type: string
    enum: [ACTIVE, DORMANT, CLOSED]
  currency:
    type: string
    pattern: '^[A-Z]{3}$'
  balances:
    $ref: '#/components/schemas/AccountBalances'
```

Money:

```
type: object
required: [amount, currency]
properties:
  amount:
```

```
type: string
pattern: '^-\?d+(\.\d{1,2})?$',
currency:
type: string
pattern: '^[\A-Z]{3}$'
```

Schema rule

Schema constraints should be machine-testable. Business rules that depend on account state, entitlements, or multiple fields may require service-layer validation in addition to JSON Schema.

5. Source-to-API Mapping

A mapping specification links the external contract to source attributes without leaking source terminology into the API. It also gives development and QA teams a common traceability artifact.

API field	Illustrative source	Transformation	Validation	Req/Test ID
accountId	Account.account_key	Opaque ID service	Required; non-empty	API-ACCT-001
status	Account.status_cd	Map A→ACTIVE, D→DORMANT, C→CLOSED	Enum membership	API-ACCT-002
currency	Account.currency_cd	Uppercase	ISO 4217 pattern	API-ACCT-003
balances.current.amount	Balance.ledger_amt	Decimal to 2 dp; string	Numeric pattern	API-BAL-001
balances.available.amount	Balance.available_amt	Decimal to 2 dp; string	Numeric pattern	API-BAL-002
openedDate	Account.open_dt	Date only	YYYY-MM-DD	API-ACCT-004

5.1 Validation Strategy

- Contract validation: type, format, required fields, max length, patterns, and enum values.
- Business validation: permitted state transitions, entitlements, cross-field rules, account status, and product constraints.
- Source validation: null handling, code translation, precision, and unexpected source values.
- Consumer contract tests: validate that additive changes do not break known clients.

5.2 Standard Error Model

```
{
  "error": {
    "code": "ACCOUNT_NOT_FOUND",
    "message": "The requested account could not be found.",
    "correlationId": "c9bd9e3f-2c62-4ae2-8c64-46d905fdf1c7",
    "details": []
  }
}
```

Error messages must not expose internal table names, stack traces, hostnames, SQL, or confidential values. `correlationId` supports supportability without disclosing implementation details.

6. Versioning & Compatibility

The API contract is treated as a product. Changes are classified by consumer impact before implementation.

Change	Impact	Version action	Control
Add optional field	Non-breaking	Same major version	Update schema + examples; notify consumers when material.
Add enum value	Potentially breaking	Assess consumer parsing	Use extensible enum strategy or coordinated release.
Rename/remove field	Breaking	New major version	Run migration window and deprecation plan.
Change field type	Breaking	New major version	Never silently change string ↔ number semantics.
Tighten validation	Potentially breaking	Impact analysis required	Check existing producer/consumer payloads first.

6.1 Security & Data Classification

- Classify each field by sensitivity (for example: Public, Internal, Confidential, Restricted) according to the organization's approved policy.
- Do not expose full account numbers, authentication secrets, government identifiers, or high-risk personal data unless the use case explicitly requires it and controls are approved.
- Apply transport encryption, authentication, authorization, least privilege, and field-level masking appropriate to the API risk profile.
- Keep sensitive values out of logs. Use correlation IDs and structured operational metadata for diagnostics.
- Document data retention and downstream caching expectations for every consumer class.

6.2 Observability

Operational telemetry should measure API health without duplicating business-sensitive payloads. Recommended dimensions include route, response class, latency, dependency result, schema-validation failure, correlationId, and deployment version.

7. Governance & Change Control

Schema governance ensures that API changes remain intentional, traceable, and testable. The review is lightweight for additive changes and more formal for compatibility or security impacts.

Stage	Primary owner	Required artifact
1. Requirement	Business Analyst / Product	Business rule, consumer need, acceptance criteria
2. Model	BA + Solution/API Design	Canonical entity/field definition, ownership, optionality
3. Contract	API Design / Development	OpenAPI + JSON Schema + examples
4. Mapping	Integration / Data	Source mapping and transformation rules
5. Review	Security / Architecture / QA	Compatibility, privacy, validation, error semantics
6. Test	Development + QA	Contract, integration, negative, regression tests
7. Release	Product / Delivery	Version, release notes, consumer communication

7.1 Definition of Done — Schema Checklist

- Business meaning documented for every field and object.
- Required / optional / nullable behavior explicitly defined.
- Identifiers, dates, timestamps, currency, and money follow standard conventions.
- Source-to-API mapping and transformations are traceable.
- Error responses and negative scenarios are modeled and tested.
- No sensitive implementation details are exposed in payloads or errors.
- Backward-compatibility impact has been assessed.
- OpenAPI / JSON Schema validates successfully in CI.
- Example payloads use fictional data and align with the schema.
- Consumer contract and regression tests are updated.

Portfolio note

This sample demonstrates structured requirements thinking, API contract design, data mapping, validation, versioning, and governance. It intentionally avoids proprietary system details and uses fictional banking data.