Module 6
Designing Effective APIs

At the end of this module, you should be able to

- Appreciate the importance of **contract-first API design** and **RAML fragments**
- Opt for **semantic API versioning** and where to expose what elements of an API's version
- Choose Enterprise **Data Model** or Bounded Context Data Models
- Design **System APIs** to abstract from backend systems
- Apply HTTP-based **asynchronous** execution of API invocations and **caching** to meet NFRs
- Identify **idempotent** HTTP methods and HTTP-native support for optimistic **concurrency**

All contents © MuleSoft Inc.

Understanding API design on Anypoint Platform

API design with Anypoint Platform and RAML

- MuleSoft advocates **API specification-driven** API design
  - As shown so far
- Anypoint Platform has support for API specifications in the form of
  - **RAML** definitions
    - First-class support in all relevant components
  - **OpenAPI** (OAS, Swagger) documents
    - Import/export in Design Center
    - Import in Exchange
  - **WSDL** documents
    - Import in Exchange

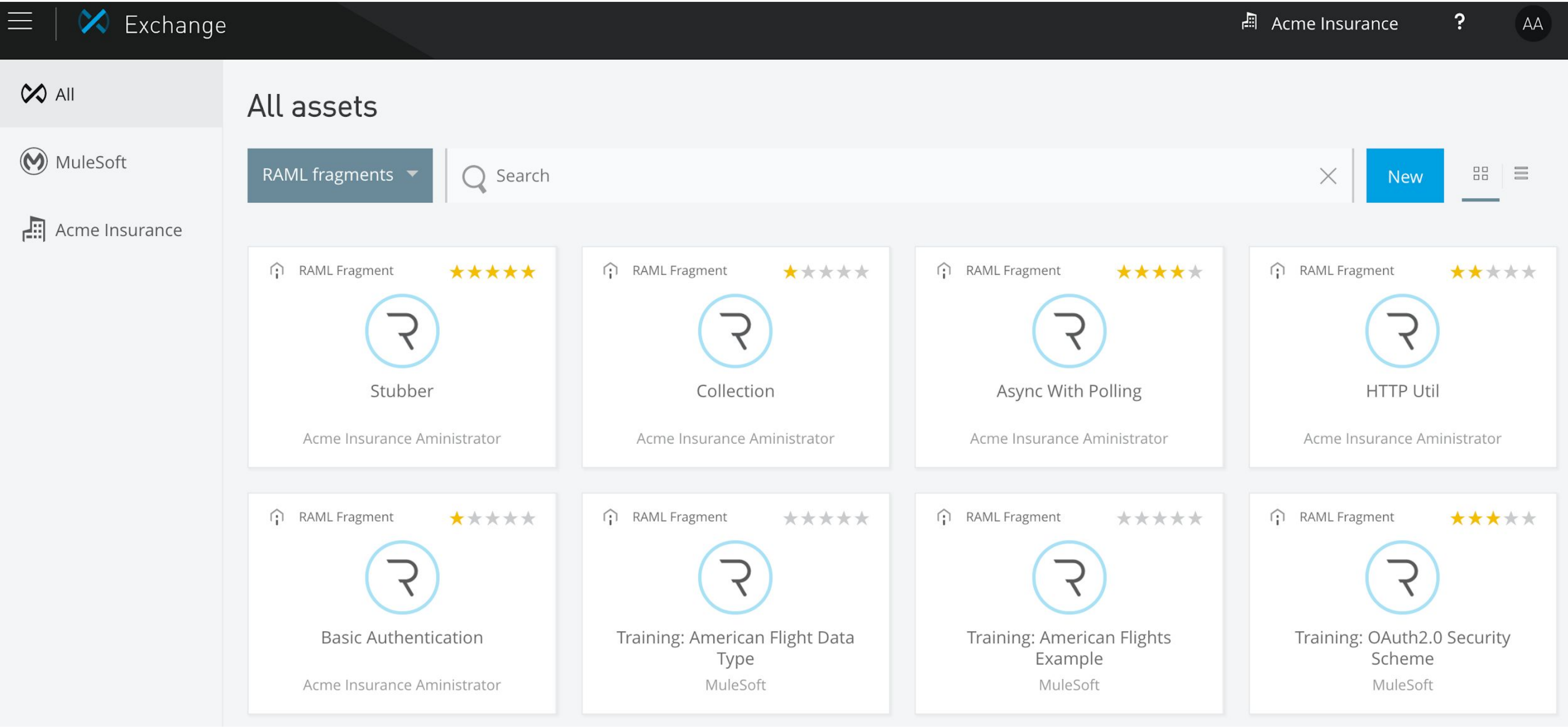
All contents © MuleSoft Inc.

Identifying and publishing reusable RAML fragments

- RAML fragments are **reusable parts** of a RAML definition
  - Partial interface definitions
- **C4E** owns discovery and publication of RAML fragments:
  - RAML **SecurityScheme** for HTTP Basic Authentication and OAuth 2.0
  - RAML Library with **resourceTypes** for **collections of items**
  - RAML Library with **resourceTypes**, traits for **asynchronous processing** of API invocations
  - RAML traits for **API policies** at Acme Insurance:
    - Client ID enforcement
    - SLA-based and non-SLA-based Rate Limiting and Throttling
- RAML fragments are represented in Anypoint Platform as
  - **Design Center projects**
  - **Exchange assets**

All contents © MuleSoft Inc.

Identifying and publishing reusable RAML fragments



Versioning APIs

Strategy for API versions on Anypoint Platform

- Follow **split strategy** on versioning APIs:
  - Try to make all **changes backwards-compatible**
  - **Version** APIs in case incompatible changes will be needed
- API versioning approach is **visible** throughout the application network
  - **Standardized by C4E**
  - API version visible in:
    - **API endpoint URL**
    - **RAML definition**: version and baseUri
    - **Exchange entry**: "API version", "Asset version", "API instances"
    - **API Manager entry**: "API version", "Asset version", "Implementation URL" and "Consumer endpoint"

All contents © MuleSoft Inc.

Understanding semantic versioning of APIs

- **Major** versions for backwards-incompatible changes
  - Require API clients to adapt
- **Minor** versions for backwards-compatible changes
  - Do not require API clients to change, unless they want to take advantage of the newly introduced changes
- **Patch** versions for small fully backwards-compatible fixes
- Version 1.2.3 of an API is a perfect stand-in for version 1.1.5

All contents © MuleSoft Inc.

Understanding semantic versioning of APIs

- **Major version** visible in
  - **API endpoint URL**
  - **RAML definition**: version and baseUri
  - **Exchange entry**: "API version"
  - **API Manager entry**: "API version", "Implementation URL", "Consumer endpoint"
- **Full semantic version** visible in
  - **Exchange entry**: "Asset version", "API instances"
  - **API Manager entry**: "Asset version"

All contents © MuleSoft Inc.

Versioning API endpoint URLs

- Just in the **URL path**
  - E.g., **http://ans-policyholdersummary-papi.us-e2.cloudhub.io/v1**
  - Requires acmeins-policyholdersummary-papi.us-e2.cloudhub.io to support all version of the API
  - Or use of version-based routing of API invocations
- Just in the **hostname**
  - E.g., **http://ans-policyholdersummary-papi-v1.us-e2.cloudhub.io/**
- **Hostname and URL path**
  - E.g., **http://ans-policyholdersummary-papi-v1.us-e2.cloudhub.io/v1**
  - Redundant but no need for URL rewriting
  - Allows same API implementation to expose endpoints for more than one major version

All contents © MuleSoft Inc.

API versioning guidelines

C4E defines:

- Only expose major API versions as **v1, v2**, etc. in **RAML definition**, API **endpoint URL** and API Manager entries in "API version", "Implementation URL" and "Consumer endpoint"
- API **endpoint URL**: http://ans-policyholdersummary-papi.us-e2.cloudhub.io/v1
  - Future major versions either implemented in same API implementation or to route API invocations based on version
  - In future, augmented with CloudHub DLB-supported URL mapping
- **Exchange versioning** as enforced by publishing to it

All contents © MuleSoft Inc.

Deciding granularity, separation and abstraction of APIs

Defining API data model

- Data types that appear in an API form the **API data model**
  - By definition part of the interface **contract** for that API
  - **Visible** across the application network
  - Specify in **RAML definition**
- Examples:
  - JSON representation of **Policy Holder** of a Motor Policy returned by "Motor Policy Holder Search SAPI"
  - XML representation of **Quote** returned by "Aggregator Quote Creation EAPI" to Aggregator
  - JSON representation of a **Motor Quote** to be created for a given Policy Holder passed to "Motor Quote PAPI"
- Do not confuse with **API implementation-internal models** for domain model, persistence, etc.

All contents © MuleSoft Inc.

Enterprise Data Model versus Bounded Context Data Models

- **Enterprise Data Model**
  - Exactly one **canonical definition** of each data type (**CDM**)
  - **Reused** in in all APIs that require that data type
  - E.g., one Acme Insurance-wide definition of **Policy**
- **Bounded Context Data Model**
  - The **set of data types** for a given Bounded Context
  - **APIs in a Bounded Context** use its Bounded Context Data Model
  - **Several Bounded Contexts** identified within Acme Insurance by their usage of common terminology and concepts
  - E.g., **Motor Claims** Bounded Context has distinct definition of **Policy**, unrelated to definition of Policy in **Home Underwriting** Bounded Context
- **Every API** could have its **own API data model**
  - In its own separate Bounded Context

All contents © MuleSoft Inc.

Selecting between Enterprise Data Model and Bounded Context Data Models

- **Coordination** of data models adds **overhead**
  - Applies to **initial data modelling**, all **changes** and **rollout** of changes
  - Can become significant if APIs are owned by **separate groups**
  - One reason why **Enterprise Data Models** often not successful
- If no successful Enterprise Data Model: **use Bounded Context Data Models**
- If there is a **successful Enterprise Data Model**
  - All **Process APIs** and **System APIs** should reuse it
  - **Experience APIs** rarely served well be an Enterprise Data Model

All contents © MuleSoft Inc.

Identifying Bounded Contexts and Bounded Context Data Models

- **Identify Bounded Contexts**
  - Start with **organizational units** with **homogenous business concepts**
    - E.g., Motor Claims, Home Claims, Motor Underwriting, Home Underwriting, CRM
  - If in doubt prefer **smaller** Bounded Contexts
  - If still in doubt put **each API** in its own Bounded Context
- **Assign** each API to **exactly one** Bounded Context
  - Based on **defining data type** of each API
  - If needed **sub-divide API**
- **Define** a Bounded Context **Data Model** for each Bounded Context
  - Based solely on **needs of APIs** in that Bounded Context
  - **Reuse** where possible in APIs of that Bounded Context
  - **Publish** as **RAML fragments** (RAML types, Library) in Design Center and Exchange
    - **C4E** owns that activity and the harvesting of data types

All contents © MuleSoft Inc.

Exercise: Identify Bounded Contexts in Acme Insurance's application network

MuleSoft

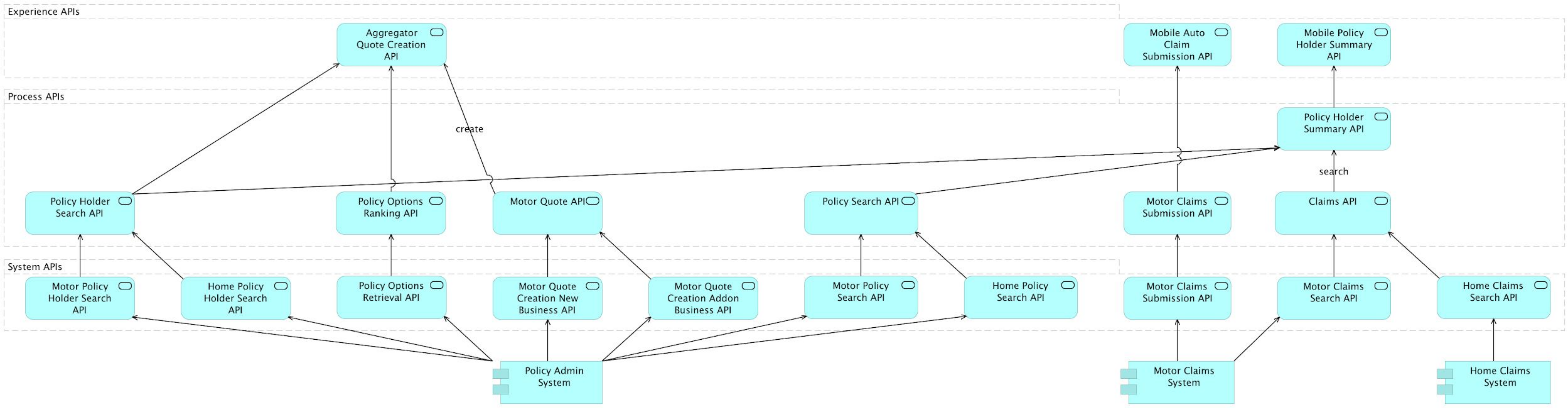
You have already identified a large number of APIs:

1. Delineate **boundaries** of meaningful Bounded Contexts so that
   - every API belongs to exactly one Bounded Context
   - more than one Bounded Context overall (i.e., no Enterprise Data Model)
2. Identify the **defining API data types** for each Bounded Context and verify that they apply to all APIs in that Bounded Context
3. Discuss the implications of the identified Bounded Context Data Models for
   - **coordination** between the teams responsible for the APIs in each Bounded Context
   - **data transformation** when APIs in different Bounded Contexts invoke each other

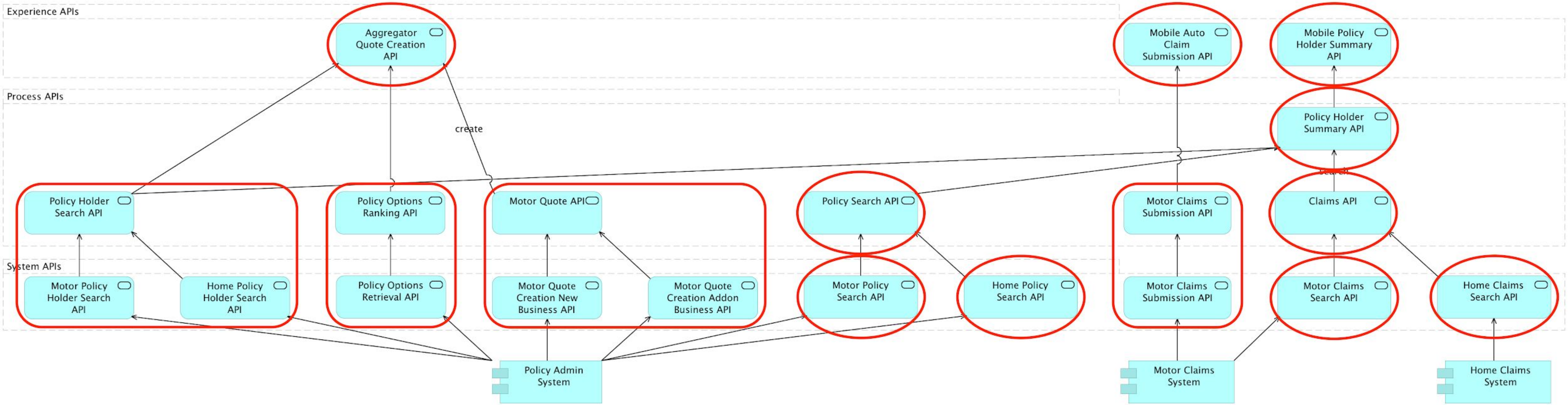
All contents © MuleSoft Inc.

Exercise: Identify Bounded Contexts in Acme Insurance's application network

1. Bounded Context **boundaries**
2. **Defining API data types**
3. Implications for
   - **coordination**
   - **data transformation**



Solution: Identify Bounded Contexts in Acme Insurance's application network



Mapping between Bounded Context Data Models

MuleSoft

- **Data transformation** whenever APIs from different Bounded Contexts need to cooperate
- E.g., **anticorruption layer**:
  - API implementation belonging to an API in Bounded Context 1
  - Invokes API in a different Bounded Context 2
  - Transform the Bounded Context Data Model 2
  - To Bounded Context Data Model 1
- A **strength** of Studio and the Mule runtime
- Not architecturally significant and hence **out-of-scope**

All contents © MuleSoft Inc.

Understanding relationships when integrating between Bounded Contexts

Power relationships whenever Bounded Contexts need to interact:

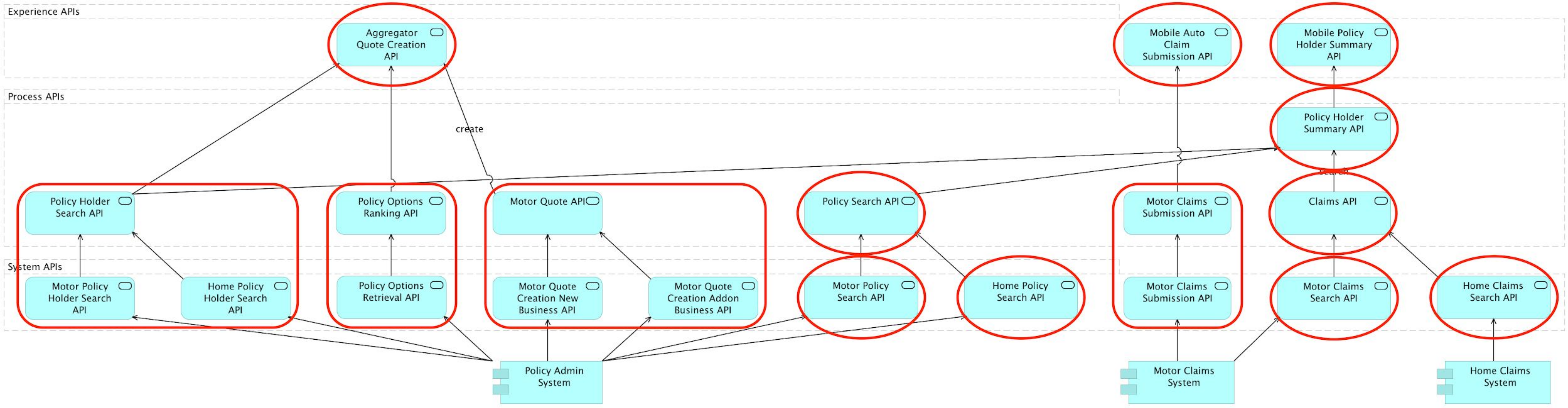
- **Partnership**
  - **Coordination** of caller and called in terms of features and timeline
- **Customer/Supplier**
  - Caller **requests** features from the called, who may have to **consolidate** many callers' feature requests
- **Conformist**
  - Caller must work with whatever called provides

All contents © MuleSoft Inc.

Exercise: Identify Bounded Context power relationships

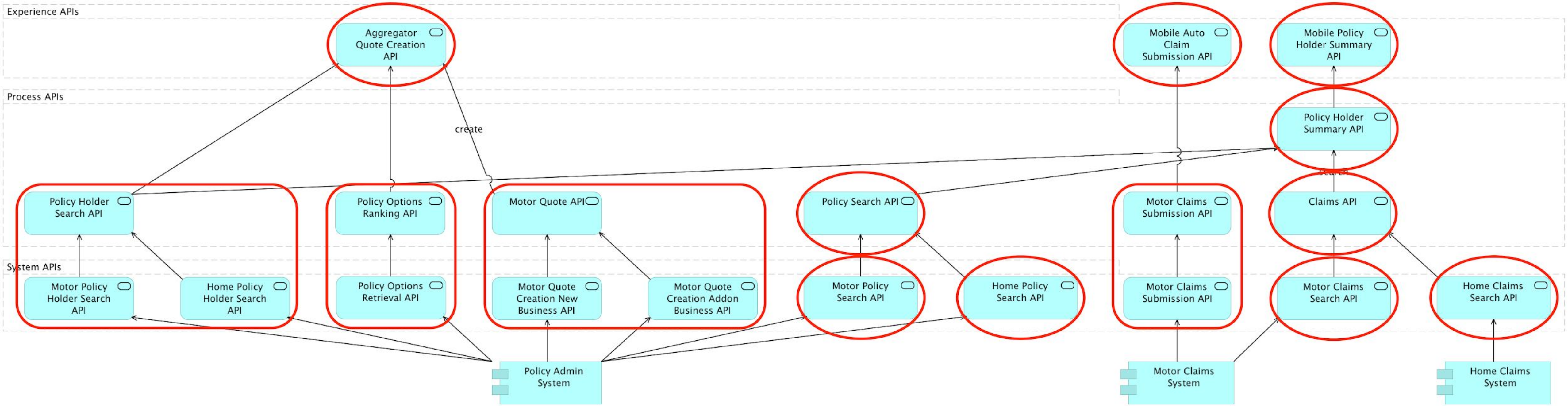
- Based on previously identified Bounded Contexts
- Or drawing on your own experience:

1. Identify at least one example for each type of relationship, i.e.,
   a. Partnership
   b. Customer/Supplier
   c. Conformist



Exercise: Identify Bounded Context power relationships

- **Partnership**:
  - Aggregator Quote Creation -> Motor Quote Creation: same LoB and product
- **Customer/Supplier**:
  - Claims -> Home Claims Search: HCS external, SAPI widely used
- **Conformist**:
  - Aggregator Quote Creation -> Aggregator



Abstracting from backend systems with System APIs

- Granularity of System APIs should make **business sense**
- If **Enterprise Data Model**:
  - System APIs use it
  - API implementations **translate** to/from backend system data model
- If **no Enterprise Data Model**:
  - **Define Bounded Context Data Model**
  - API implementations **translate** to/from backend system data model
  - Bounded Context Data Model defined by business characteristics not backend system
    - **Translation effort** may be significant

All contents © MuleSoft Inc.

Abstracting from backend systems with System APIs

- If **neither Enterprise nor Bounded Context Data Model**:
  - System APIs approximately **mirror backend system data model**
  - **Same semantics** and naming as backend system
  - Only **data types** actually needed
    - Backend system often are Big Balls of Mud
  - Lightly **sanitized**
    - Idiomatic JSON, correcting misspellings, …
  - Only **fields** actually needed
  - Idiomatic **REST**
  - **Unsatisfactory isolation** from backend systems
    - **No "swap out"** a backend system without changing System APIs
    - Isolates API clients from **protocol**, authentication, network address, …
    - Very **pragmatic**
    - **API policies**, RAML-defined **contract**
    - **Further isolation in Process API** implementations

All contents © MuleSoft Inc.

Designing Acme Insurance's System APIs

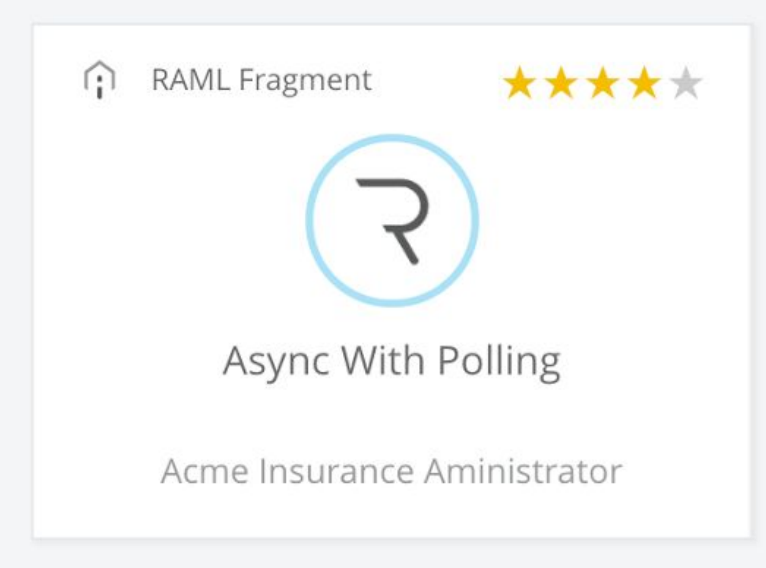
- **Many System APIs** infront of Policy Admin System, Motor Claims System and Home Claims System
  - Defined by **Bounded Contexts** (motor, home) as well as by **use cases** (policy holder search, option retrieval, etc.)
- All System APIs are **JSON REST APIs**
  - Define JSON-compatible data types in **RAML definitions/libraries**
- No Enterprise Data Model, use **Bounded Context Data Models**
  - "Motor Policy" returned by the "Motor Policy Search SAPI" **not directly related to** data definition in the **Policy Admin System**
  - **Names** with business meaning
  - **JSON** data types and JSON-compatible naming
  - **Omits fields** not relevant for Motor Underwriting

All contents © MuleSoft Inc.

Selecting API invocation patterns that address NFRs

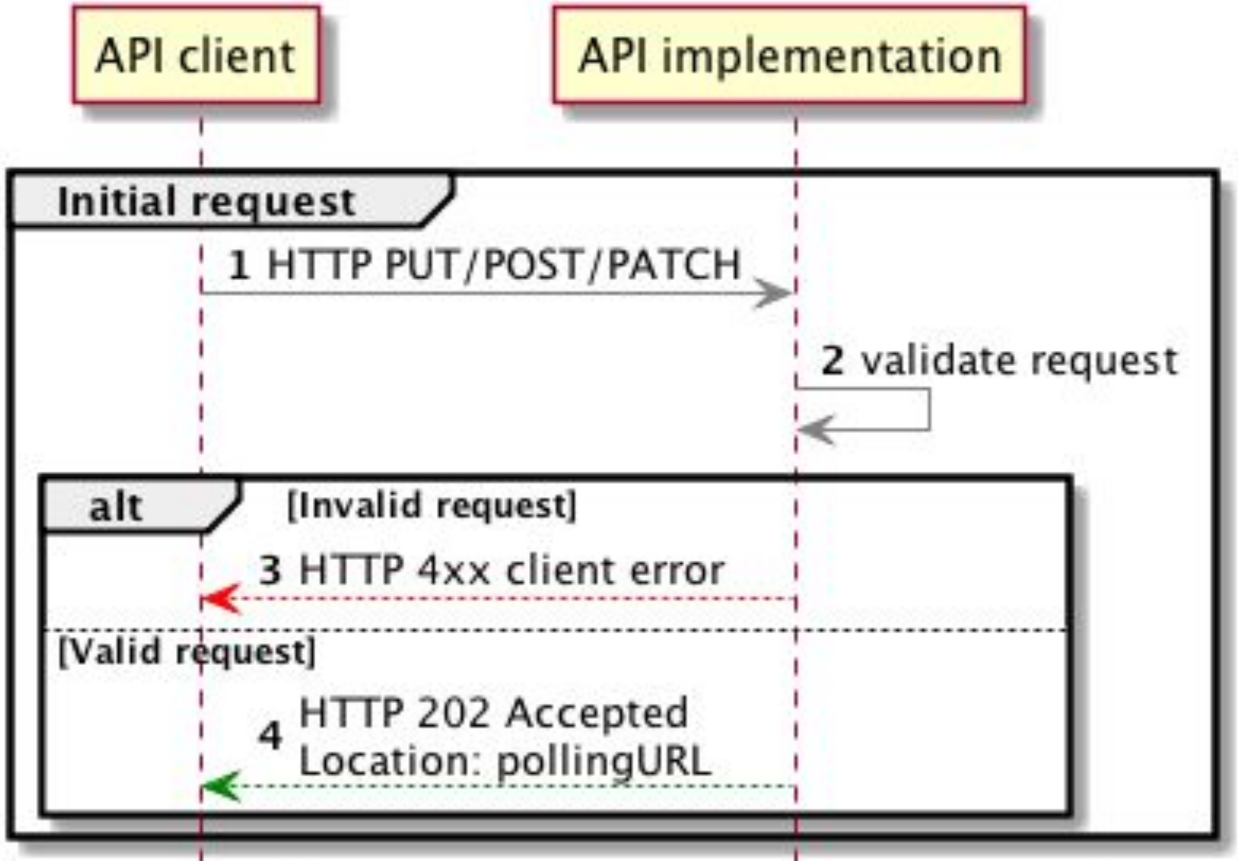
Asynchronously executing API invocations with polling

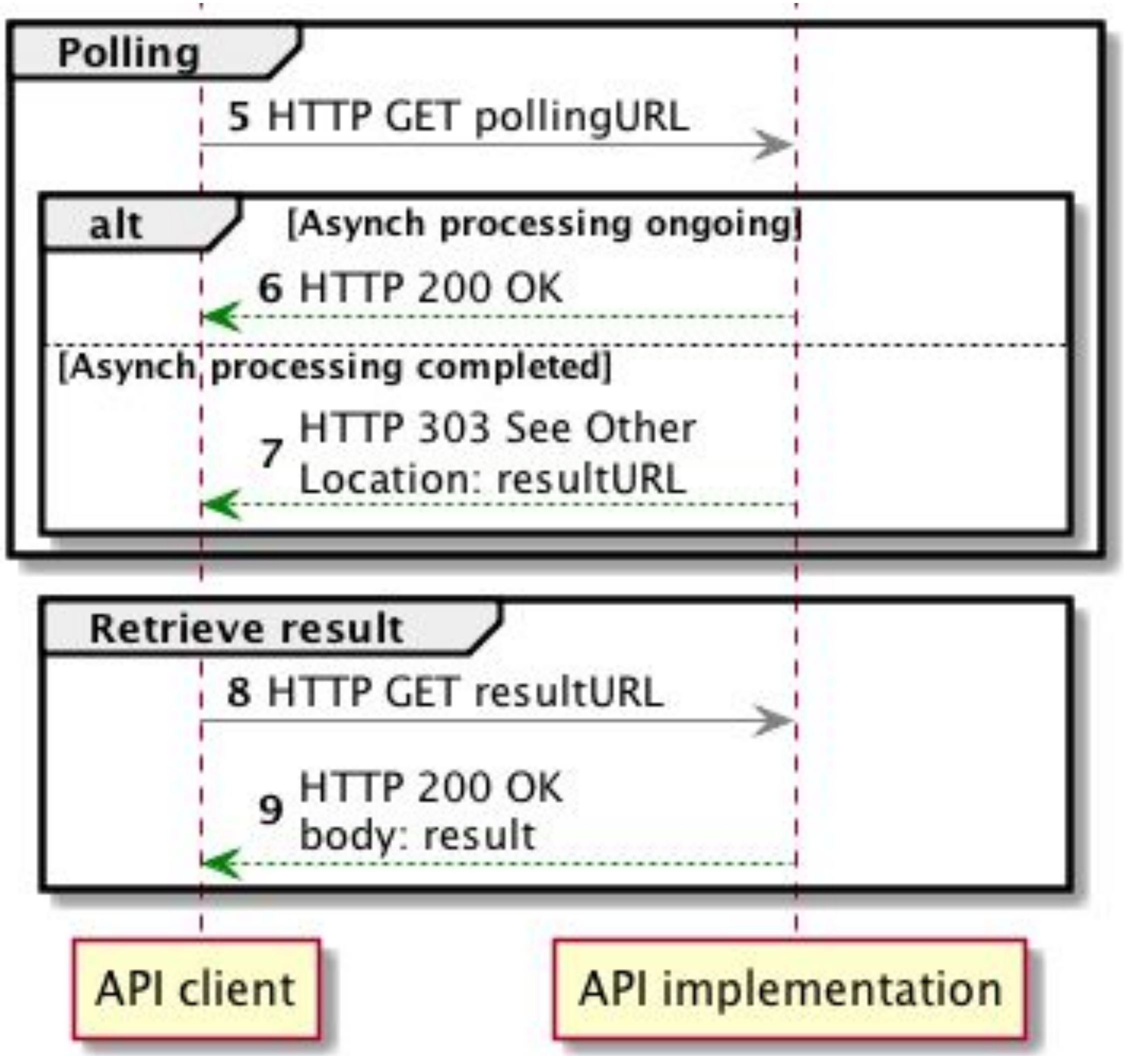
- "Submit auto claim" must be executed **asynchronously**
- HTTP has **native support** for asynchronous processing
  - Immediately available to **REST APIs**
  - But not non-REST APIs like **SOAP APIs**
- Documented in the **RAML definition** of the respective APIs
  - C4E has published **RAML library**
- Applicable to invocation of "Mobile Auto Claim Submission EAPI" by Customer Self-Service Mobile App



All contents © MuleSoft Inc.

Asynchronously executing API invocations with polling





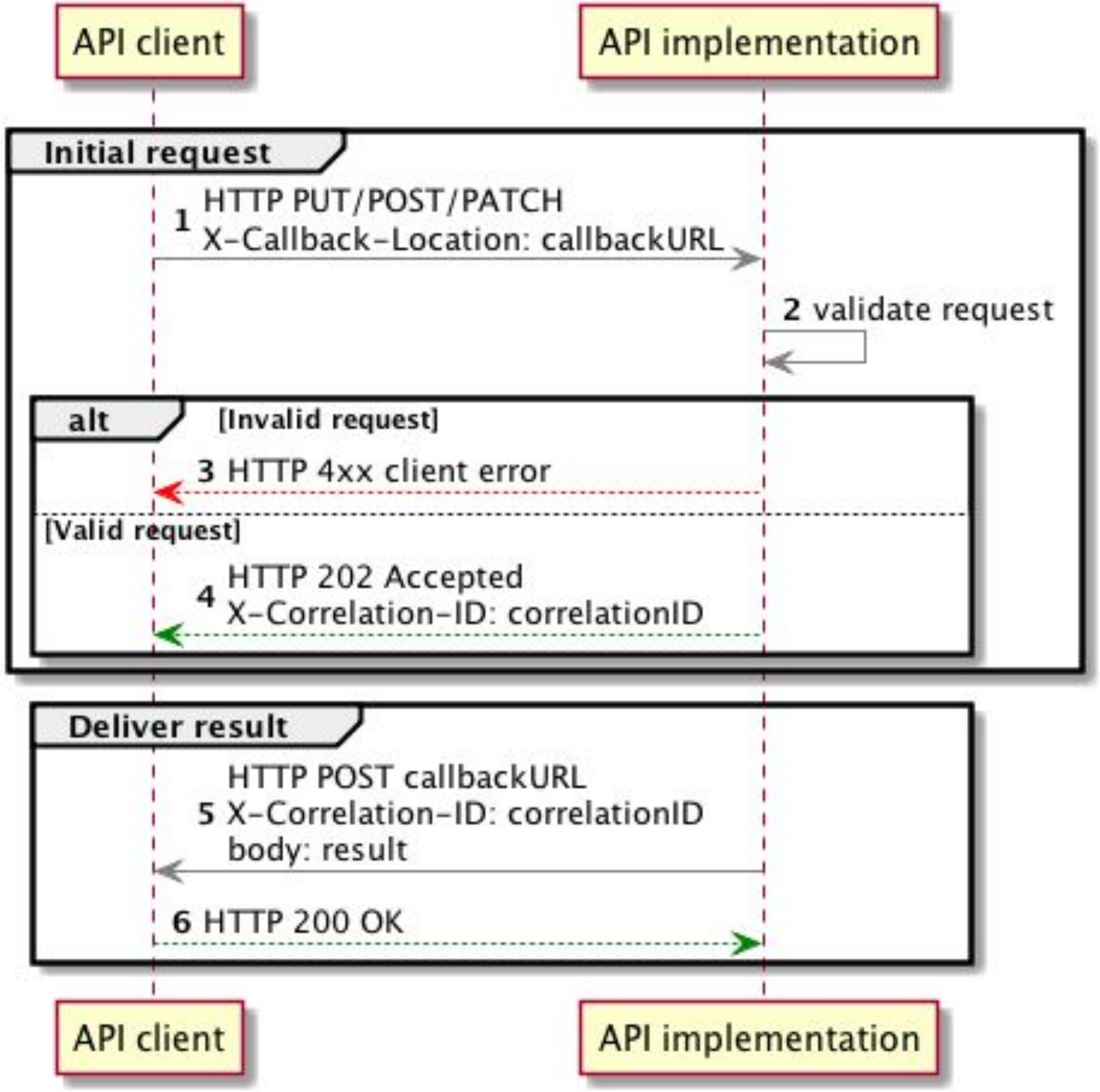
All contents © MuleSoft Inc.

Asynchronously executing API invocations with callbacks

- **Alternative** to polling is use of HTTP callbacks
  - aka **webhooks**
- Requires the original API client to be **reachable** by HTTP requests from the API implementation
- Documented in the **RAML definition** of the respective APIs
  - C4E to publish **RAML library**
- Applicable to invocation of "Motor Claims Submission PAPI" by "Mobile Auto Claim Submission EAPI"

All contents © MuleSoft Inc.

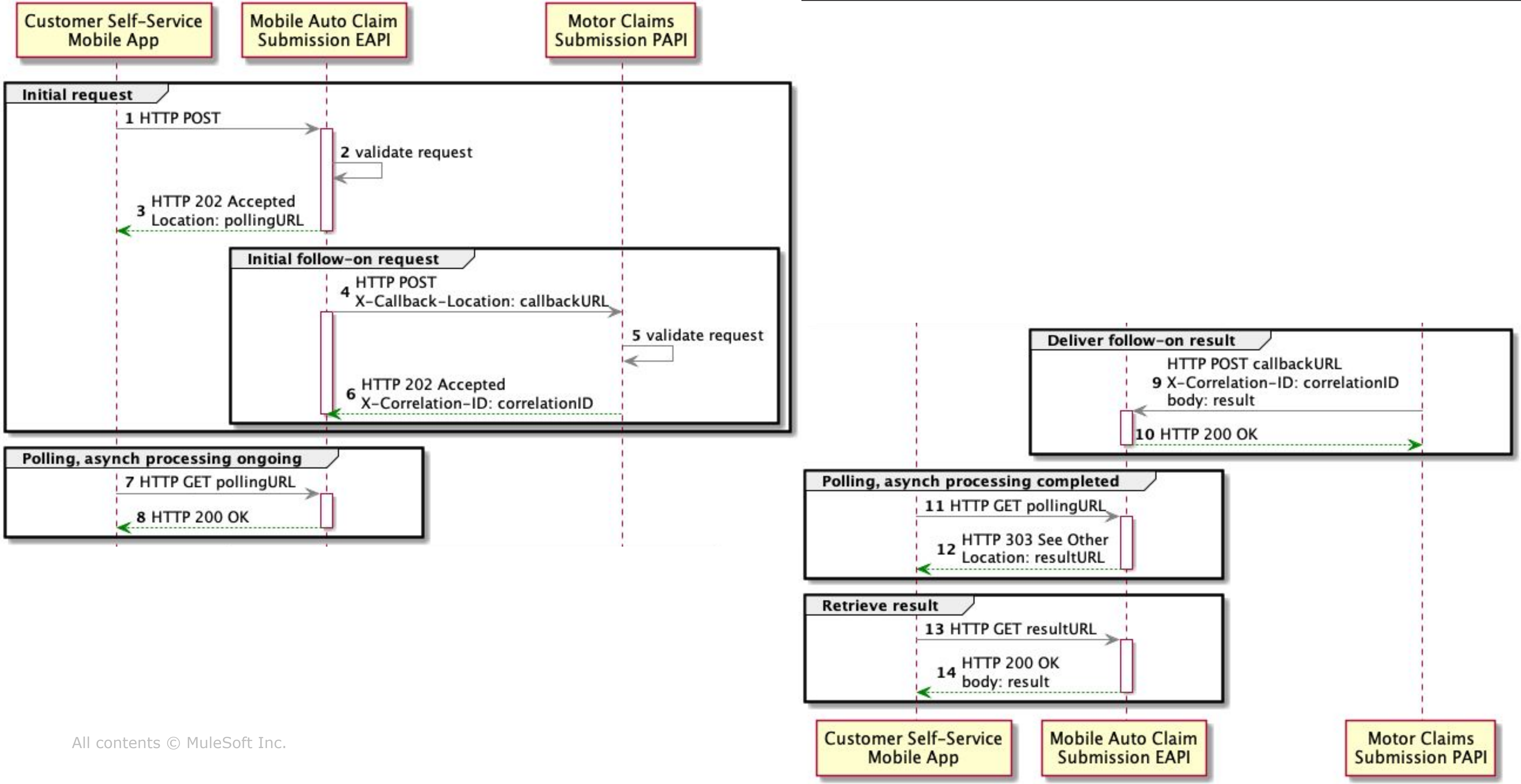
Asynchronously executing API invocations with callbacks



All contents © MuleSoft Inc.

Asynchronous API invocations for the "Submit auto claim" feature

MuleSoft



All contents © MuleSoft Inc.

State handling for asynchronous execution of API invocations

- Asynchronously executing API invocations requires API implementations to **maintain state** of each async invocation
  - Makes API implementations **stateful**
  - or delegate state management to **stateful component**
- Options in **Mule apps**:
  - Arbitrary state:
    - **Object Store**, a feature of the Mule runtime
    - External **databases**
  - Messages:
    - **VM queues**
    - External **message brokers**

All contents © MuleSoft Inc.

Exercise: Meaning and implications of statefulness

Asynchronous execution of claim submissions is the first example in our Enterprise Architecture of stateful API implementations:

1. Discuss the exact meaning of "stateful API implementation"
2. Is statefulness of the API implementations of the "Mobile Auto Claim Submission EAPI" and the "Motor Claims Submission PAPI" avoidable?
3. List scenarios where statefulness makes a difference

All contents © MuleSoft Inc.

Solution: Meaning and implications of statefulness

- **Stateful**: API implementation **nodes store data (RAM, disk)**
  - **Not** when storing in **external database** or message broker
  - **Local** to a node or **replicated** to all nodes
- **State handling** for these APIs **is unavoidable** - **statefulness** of API impls is **architectural choice**
  - **CloudHub Object Store** stores state outside nodes
  - State in local memory/disk or in-memory replicated amongst nodes (options with Object Stores, not all available in CloudHub): **stateful** API impls
- Makes a difference for:
  - **Load-balancing**
  - **Updates/restarts** and **relocation**
  - **Scalability**
  - **Latency**
  - **Operations**, incl. data purging

All contents © MuleSoft Inc.

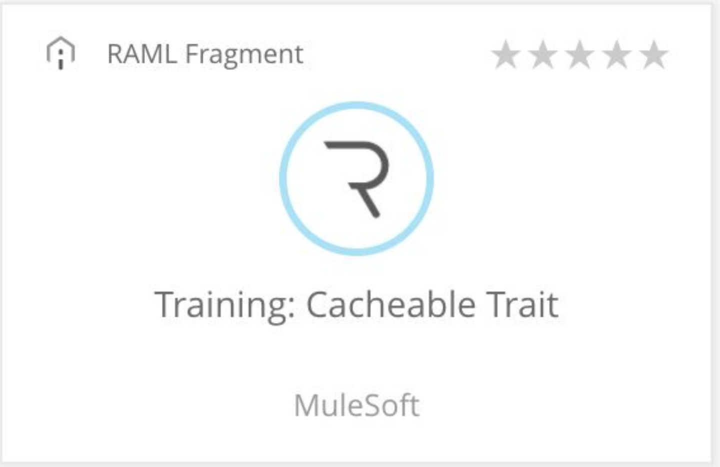
Caching and safe API methods

- HTTP standard requires these HTTP methods on any resource to be **safe**:
  - **GET**
  - **HEAD**
  - **OPTIONS**
- Safe methods **must not alter state** of underlying resource
- HTTP responses to requests using safe methods **may be cached**
- Responsibility of **API implementations** that safe methods actually never change the state of a resource

All contents © MuleSoft Inc.

Caching API invocations using HTTP facilities

- Only **safe API methods** may return a cached HTTP response
- Caching may **occur in**
  - API **client**
  - API **implementation**
  - Caching **proxy**, API **policy**, …
- Caching **requires**
  - **Storage** management
  - Manipulation of **HTTP headers** in accordance with HTTP spec
    - Cache-Control, Last-Modified, Age
    - ETag, If-Match, If-None-Match, If-Modified-Since
- Document in the **RAML** definition
- Mule apps can use **caching scope**
- **HTTP Caching API policy**

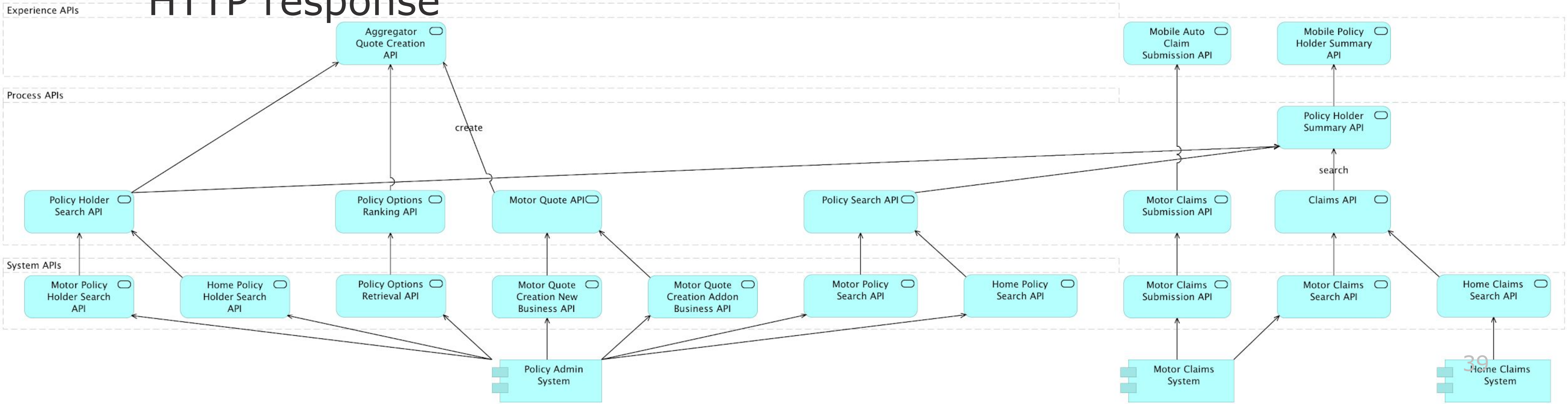


All contents © MuleSoft Inc.

Exercise: Apply caching to API invocations

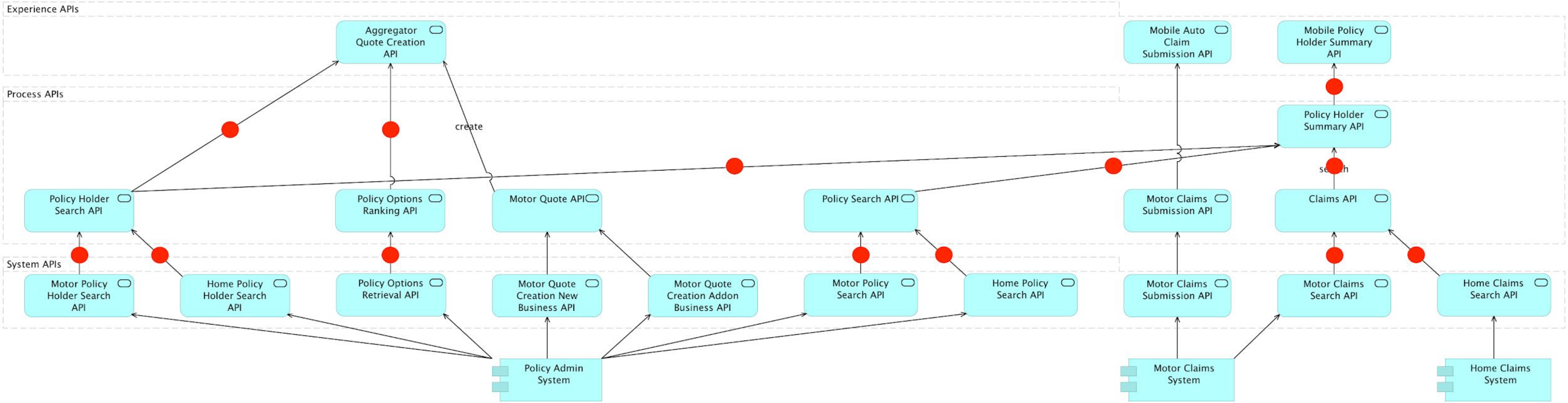
MuleSoft

1. Which API invocations are cacheable (safe) and which are not?
2. Is caching likely to benefit performance of the former?
3. Where is caching best performed?
4. Implications of caching site for storage mgmt and deployment?
5. Identify cache parameters, i.e., criteria for returning a cached HTTP response



Solution: Apply caching to API invocations

1. **Benefit** of caching: **throughput, hit rate, processing time**
2. **Client-side** caching avoids network roundtrip but reduces hit rate
3. Cache **storage** best in RAM: worker size, local-only, hit rate
4. Lookup by **API "input"** not entire HTTP request, select TTL, size with business knowledge



Retrying and idempotent API methods

- HTTP standard requires these HTTP methods on any resource to be **idempotent**:
  - **GET, HEAD, OPTIONS** (also safe/cacheable)
  - **PUT, DELETE** (unsafe/not cacheable)
- Idempotent methods do not cause **duplicate processing** when the HTTP request is re-sent
- HTTP requests using idempotent methods **may be retried** upon failure
- Responsibility of **API implementations** that PUT and DELETE actually do not cause duplicate processing
  - Mule 3 apps can use **idempotent-message-filter**

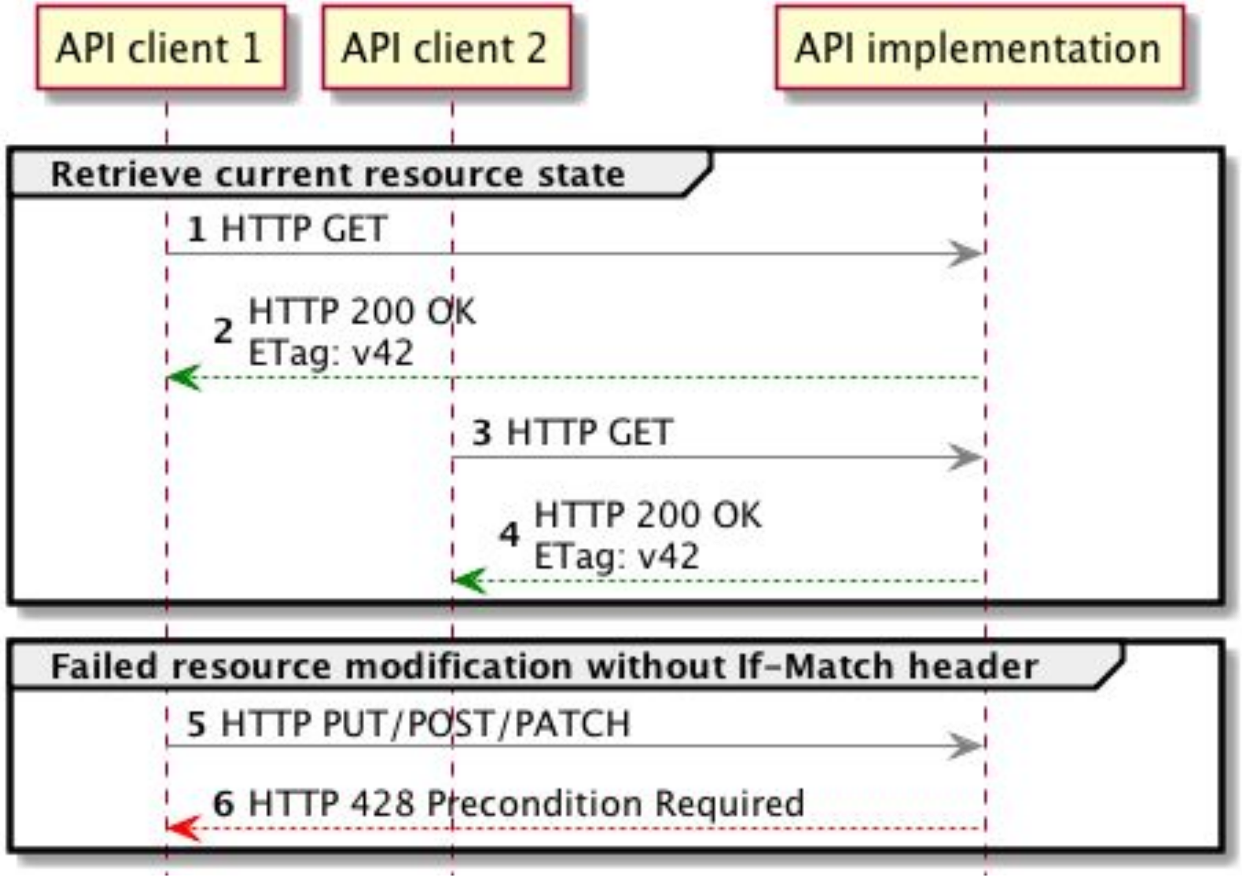
All contents © MuleSoft Inc.

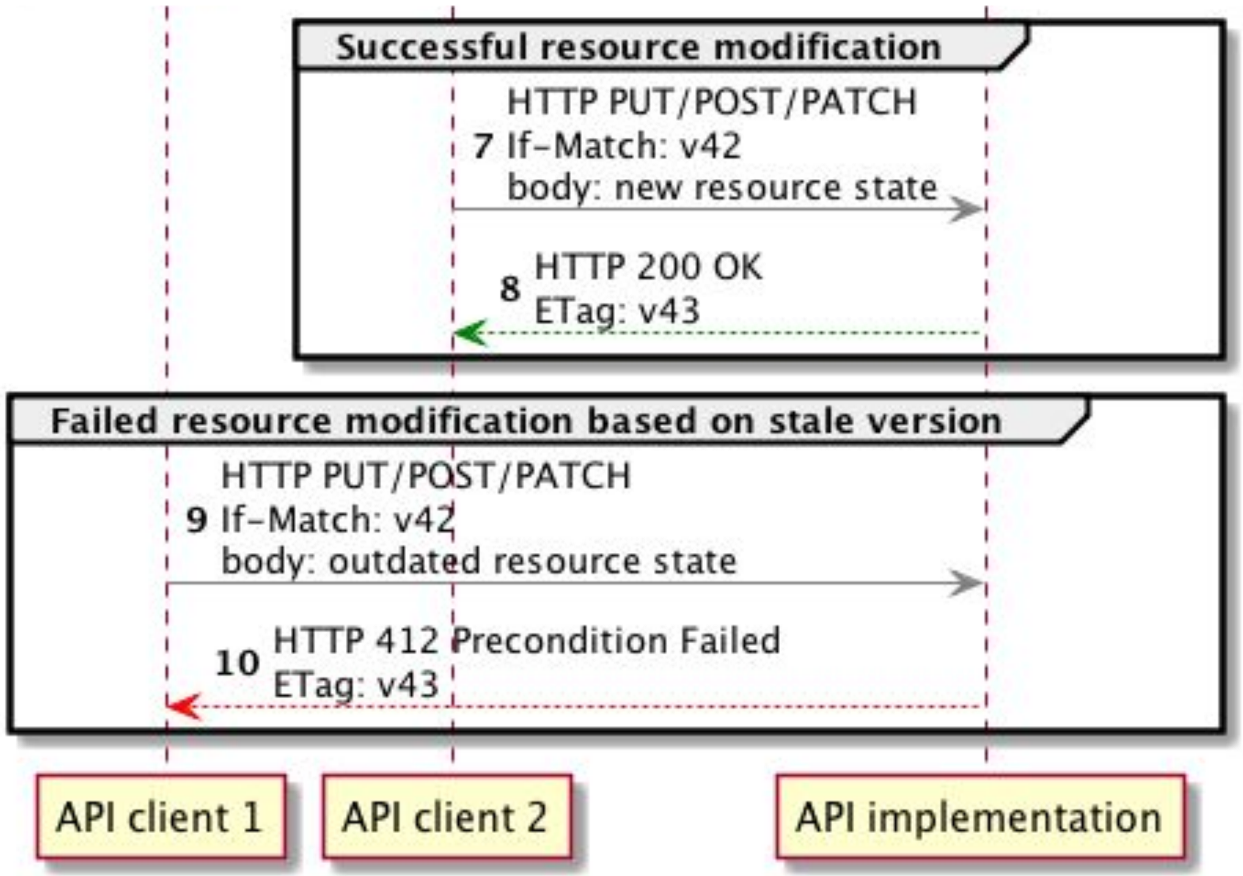
HTTP-based optimistic concurrency control

- Updates to some resources must be **serialized**
- HTTP natively supports **optimistic concurrency** control using
  - ETag, If-Match
  - HTTP 412 Precondition Failed
  - HTTP 428 Precondition Required
- Changes API's **contract**
  - Document in **RAML** definition
  - Define reusable RAML fragment (trait) and publish to
    - Design Center
    - Exchange

All contents © MuleSoft Inc.

Protecting resources from concurrent modification



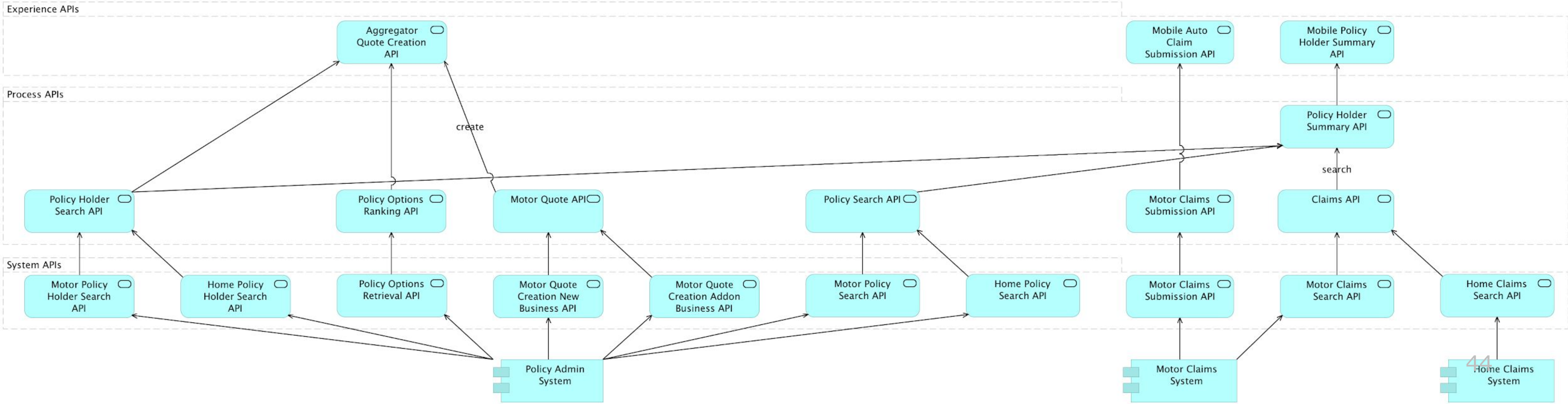


All contents © MuleSoft Inc.

Exercise: Identify API resources that may require protection from concurrent modification

Inspect all APIs in Acme Insurance's application network:

1. Identify resources that may require protection from concurrent modification
2. Discuss what it would mean for their API clients
3. Is this a standard feature of APIs that is generally useful?



Exercise: Identify API resources that may require protection from concurrent modification

- **No features** require or benefit from concurrency control:
  - "Create quote for aggregators" and "Submit auto claim" are not modifications but **de-novo creations**
  - "Retrieve policy holder summary" feature is **read-only operation**
- Hypothetical feature: **modification of policy holder details**:
  - Updates to details of the same policy holder might occur concurrently
  - Unlikely, **no clear business need** or value in protecting against it
- HTTP-based concurrency control to be **used with caution** and only when there is clear business value in doing so
  - Most often, better to **embrace concurrency**

All contents © MuleSoft Inc.

Summary

Summary

- Start with the **RAML definitions** and **simulate** API interactions
- Publish **reusable RAML fragments** to Exchange
- **Semantic versioning**-based API versioning strategy was chosen
  - Exposes **major version numbers** in most places
- API data models were defined by the APIs' **Bounded Context**
- **System APIs abstract** from backend systems to their Bounded Context
- HTTP-based **asynchronous execution** of API invocations and **caching** were employed to meet NFRs
- Most decisions change the **interface contract** of APIs
  - Captured in **RAML fragments** and RAML definitions
  - **Published** to Exchange

All contents © MuleSoft Inc.