Service API and Contract Design with REST Services and Microservices

hungdn@ptit.edu.vn

1. Service Model Design Considerations

Entity Service Design

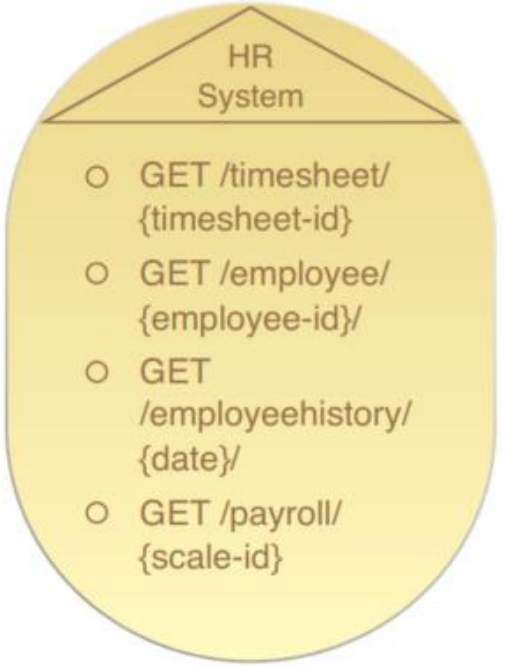
- **REST Entity Services defines a functional boundary for business entities**
  - Primarily handles **data processing** related to the entity
  - Uses **idempotent** HTTP methods (GET, PUT, DELETE) for reliability
  - **Complex Methods (Optional):** If allowed by service inventory's design standards

- **Example**
  - Invoice entity service with two standard HTTP methods and two complex methods

Utility Service Design

- **Utility Services**
  - Reusable & agnostic, but without a fixed functional scope
  - Validate method-resource choices before finalizing
- **Characteristics**
  - Flexible boundaries, often wrapping legacy systems
  - Provides **read-only** access to legacy data
- **SOA Patterns**
  - Dual Protocols
  - Concurrent Contracts
  - Service Façade
  - Legacy Wrapper

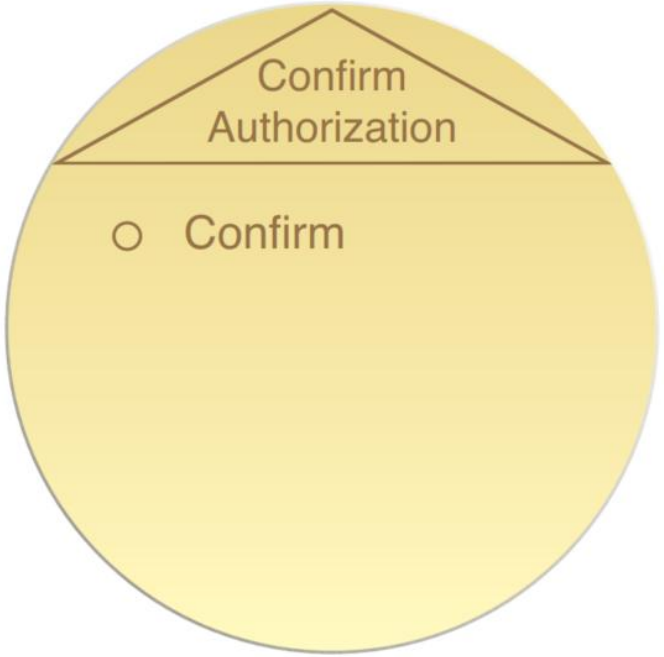
- **Example: HR System Utility Service**
  - Retrieves employee data from a legacy HR system
  - Used by other services for employee info



Microservice Design

- **Microservice**
  - Non-agnostic – Designed for specific business processes
  - Limited consumers – Often serves only one service
  - Flexibility in contract design – Standardization is optional
- **Challenges**
  - Ensuring performance & reliability and runtime efficiency
- **SOA & Microservice Patterns**
  - Microservice Deployment
  - Containerization
  - Service Data Replication
  - Redundant Implementation

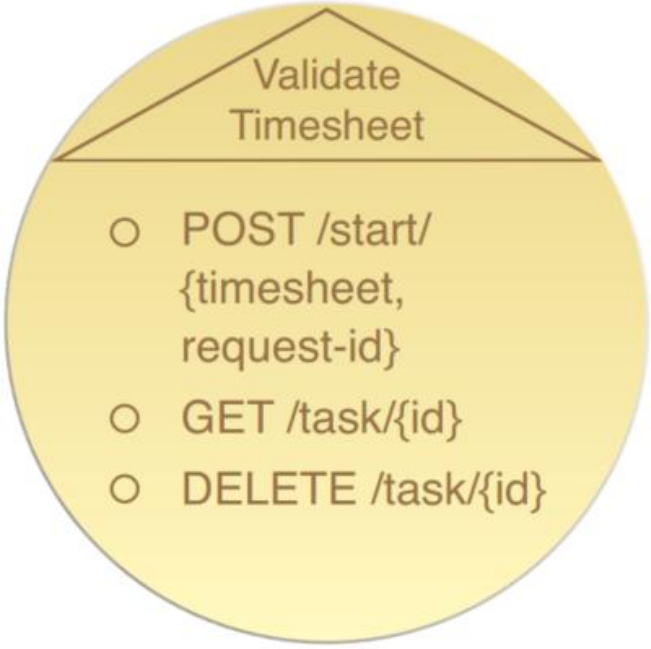
- **Example: Confirm Authorization**
  - A microservice contract with a singlepurpose, non-agnostic functional scope



Task Service Design

- **Task Service Design in REST APIs**
  - **Single-purpose** – Often limited to one key function
  - **Triggered by POST requests** – Requires reliability measures
- **Handling Asynchronous Processes**
  - For long-running tasks
  - State tracking capabilities – Using additional service endpoints
  - Cancellation support – Allows stopping active processes

- **Example: Validate Timesheet Task Service**
  - Trigger
  - Check Status
  - Cancel Task

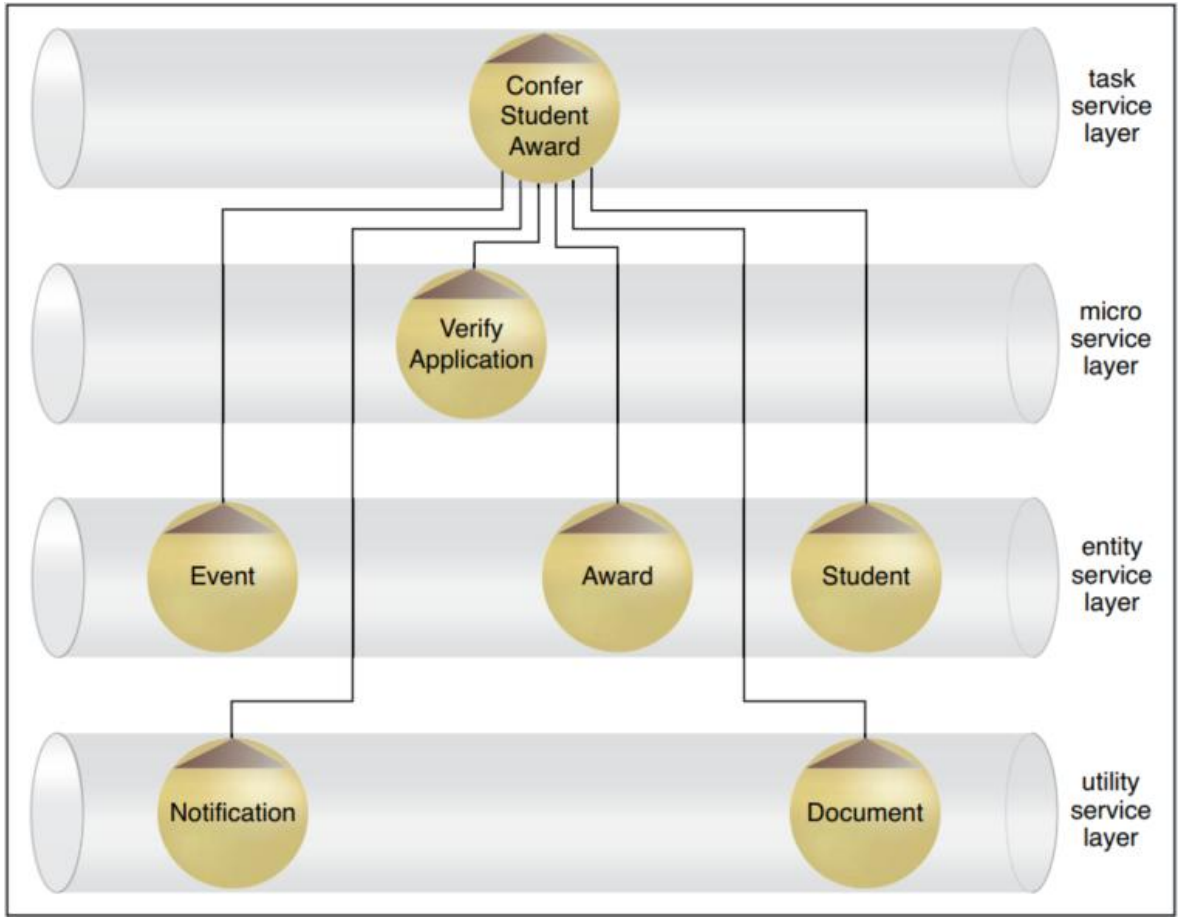


2. CASE STUDY EXAMPLE

MUA Confer Student Award

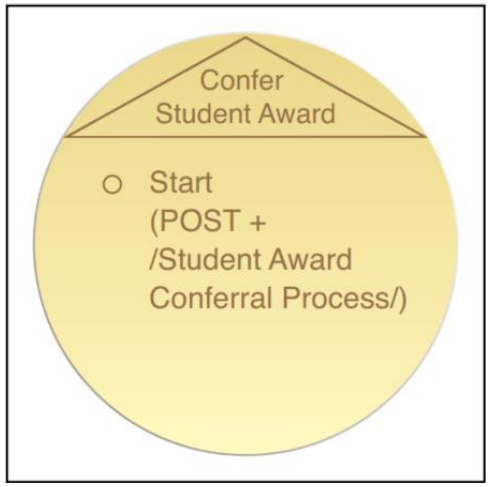
- The service modeling process performed by MUA produced a number of REST service candidates

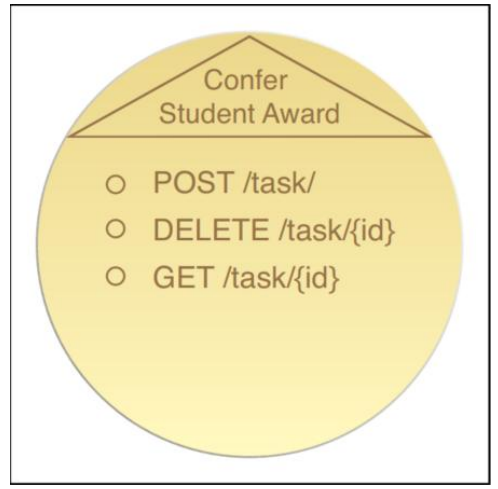
- Architects use these select service candidates (modeled in Chapter 7) as the basis for their service contract designs.



MUA Confer Student Award Service Contract

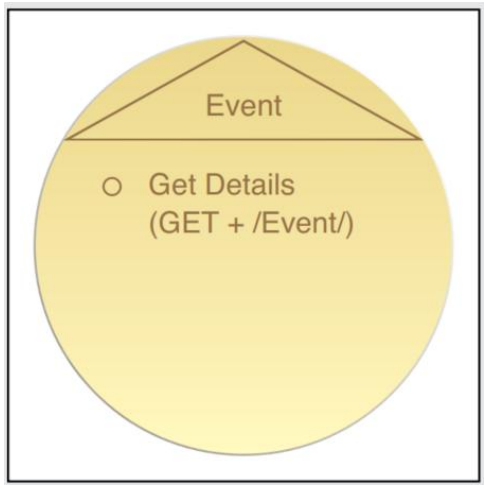
- **Student Award Application Process**
  - Students submit applications via a web form.
  - Server processes form into an XML document: `application/vnd.edu.mua.student-award-conferral-application+xhtml+xml`
- **Add new service capabilities: DELETE and GET**
- **Design Considerations**
  - Data includes **both human-readable & machine-readable** formats
  - RESTful security ensures only authorized access to sensitive data

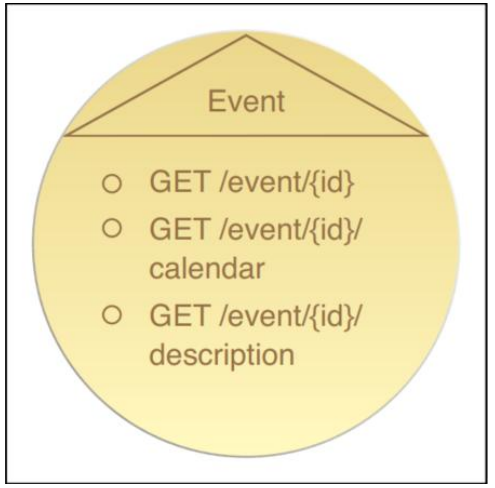




Event Service Contract (Entity)

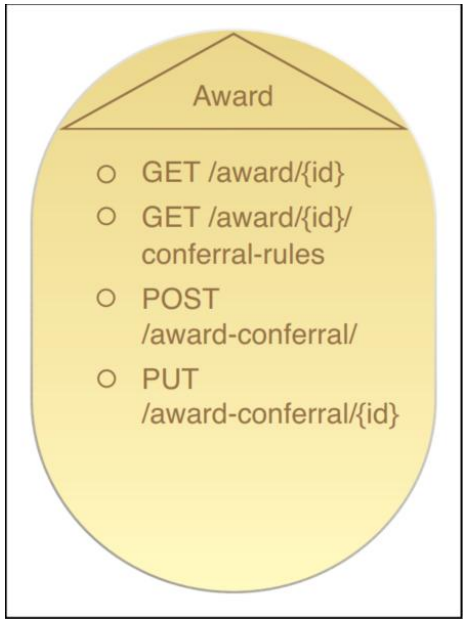
- **Added additional capabilities for broader reuse beyond the Confer Student Award process**
- **Key Capabilities**
  - **GET /event/{id}** – Retrieves event details
  - **GET /event/{id}/calendar** – Fetches event calendar details
  - **GET /event/{id}/description** – Provides a detailed event description





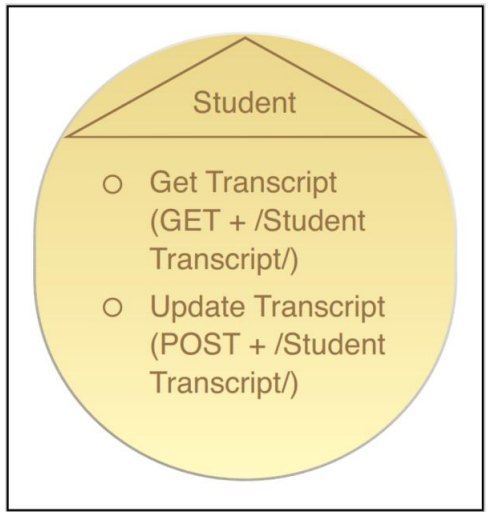
Award Service Contract (Entity)

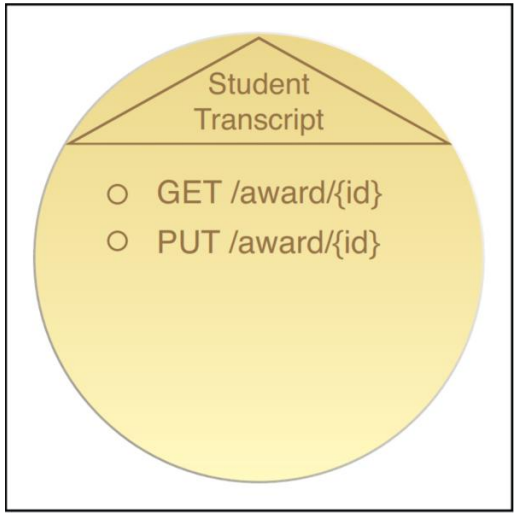
- **Uses code-on-demand approach to execute logic within the Confer Student Award Task Service**
- **GET /award/conferral-rules – Provides award rules in two formats**
  - **Human-readable HTML** (for students)
  - **Javascript format** (for integration with the task service)



Student Transcript Service Contract (Entity)

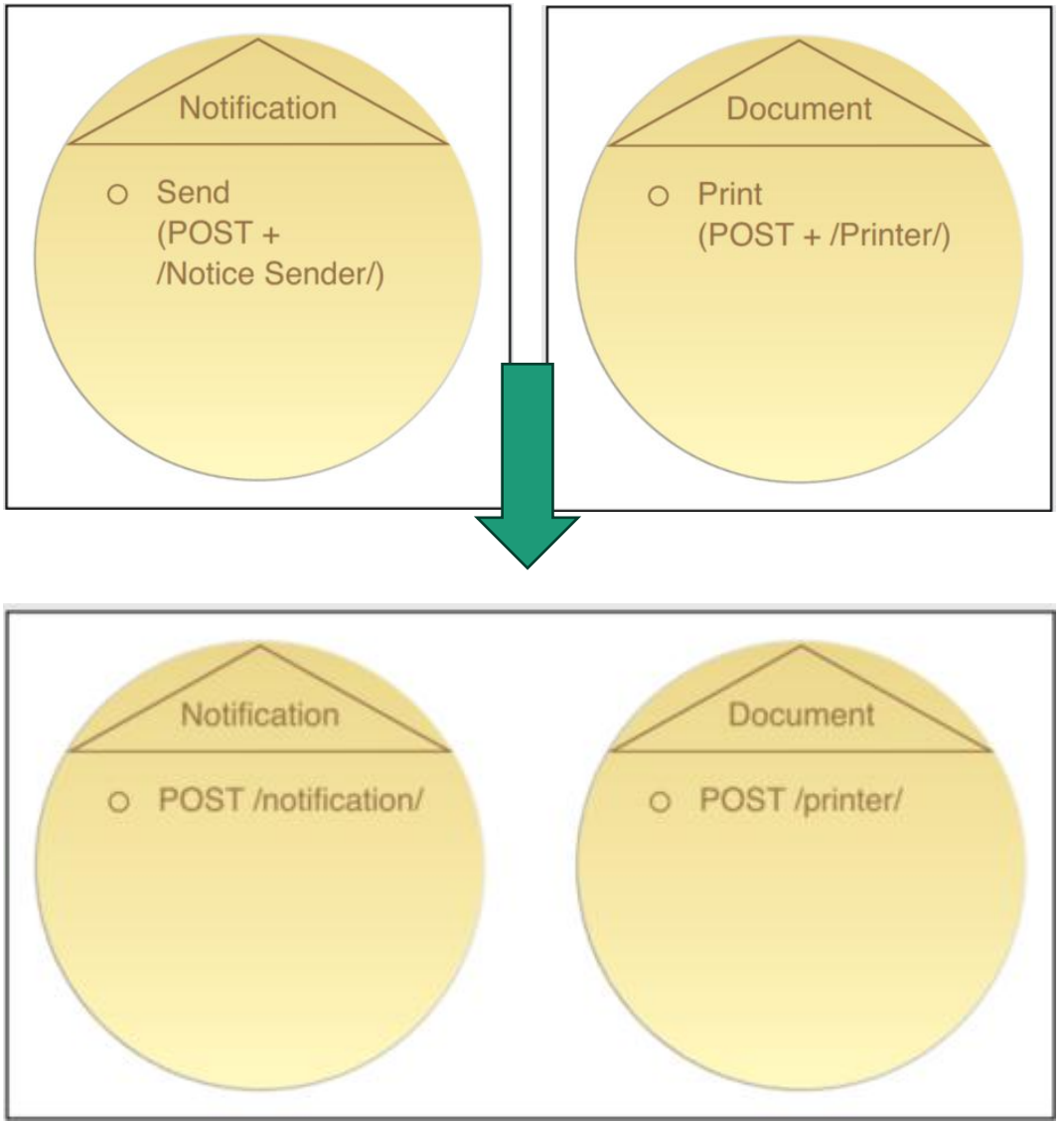
- **Why a Separate Student Transcript Service?**
  - The original **Student Service** was too broad and complex
  - It was split into **specialized entity services**, including the **Student Transcript Service**
- **Design Considerations**
  - The **Confer Student Award** process **only needs** transcript data, not full student details.
  - The **Student Transcript Service** replaces the **Student Service** in this process





Notification & Document Service Contracts (Utility)

- **Both handle human-readable data (HTML/PDF) -> Why Separate Notification & Document Services?**
  - **Notification Service** → Manages **email** notifications.
  - **Document Service** → Manages **printed** and **postal** document delivery
- **Confer Student Award process**
  - The **task service** selects the **preferred** delivery method from the application form.
  - Uses **POST** request to send notifications/documents



2. REST Service Design Guidelines

Uniform Contract & Methods

- **Key Considerations for a Uniform Contract**
  - Balance features & limitations
  - Use **web-centric technology** as a foundation but customize when needed.
  - Standardize **methods, media types, and exception handling**.
- **Designing and Standardizing Methods**
  - **Minimize methods** but allow for expansion when necessary
  - HTTP provides fundamental methods: **GET, PUT, DELETE, POST**.
    - Less common methods like **PATCH** can be used for partial updates
  - Custom protocol, methods & headers

Designing and Standardizing HTTP Headers

- **HTTP Headers in REST Services**
  - **Enhance message exchange** by providing metadata.
  - Help **service agents** process messages effectively.
  - Support **service composition** for automating business tasks.
- **Some use case for Built-in HTTP Headers**
  - Alternative to query strings
  - Response metadata
  - General service/consumer info
- **Custom Headers & Must-Understand Semantics**
  - Custom headers can **extend functionality** while maintaining backward compatibility.
  - If services **must understand** a header, a new HTTP **method** may be required
- **SOA Governance must ensure consistency in custom headers across the service inventory**

Designing and Standardizing HTTP Response Codes

- The codes and reasons provided by HTTP are standardized
- HTTP Response Code Ranges & Their Meaning
  - 100-199 → Informational
  - 200-299 → Success
  - 300-399 → Redirection
  - 400-499 → Client Errors
  - 500-599 → Server Errors

- But, HTTP does not define how consumers should react to codes → needs standardization
- SOA projects must enforce consistent response code handling

- Standardized HTTP response codes ensure predictable, efficient REST API behavior

Customizing Response Codes

- **The HTTP specification allows for extensions to response codes**
  - Custom codes **should follow existing HTTP ranges**

- **Thoughtful customization improves REST API clarity & consistency while maintaining compatibility!**

- **Guideline for Custom Response Codes**
  - Keep codes uniform
  - Make response messages human-readable
  - Ensure reusability
  - Avoid code conflicts
  - Introduce codes only when necessary

Designing Media Types

- **Importance of Media Types in Service Inventories**
  - Media types change more frequently than methods
- **Standard Web Media Types for Service Inventories**
  - text/plain; charset=utf-8
  - application/xhtml+xml
  - application/json
  - text/uri-list
  - application/atom+xml
- **Guideline for Defining Media Types**
  - Use existing media types when possible
  - Media types should be schema-specific
  - Keep them abstract
  - Enable extensibility
  - Provide standard processing instructions

Designing Media Types (2)

- Custom Media Type Naming Convention
  - Format**:** application/vnd.organization.type+supertype
- Example:
  - application/vnd.com.examplebooks.purchase-order+xml
  - **application** → Machine-readable format.
  - vnd.com.examplebooks → Vendor namespace (examplebooks.com)
  - **purchase-order** → Media type name.
  - **+xml** → Derived from XML (compatible with XML parsers).

- Thoughtful media type design enhances compatibility, extensibility, and machine readability in RESTful services.

Designing Schemas for Media Types

- **Schemas define structured data models for RESTful services.**
- **Common schema formats**
  - **XML Schema (XSD)** → Used for structured, hierarchical data.
  - **JSON Schema** → More lightweight, widely used in modern APIs.
- **Schemas must be defined at the uniform contract level → Service-specific schemas violate REST principles.**
- **Guideline, flexibility is key**
  - Schema should allow broad applicability across multiple services
  - Coarse validation constraints enable adaptability
  - Avoid overly strict typing → Helps with reusability

Complex Method Design

- **Why Customize Beyond Basic HTTP Methods?**
  - **Service inventories are controlled environments** → Can enforce stricter rules.
  - **Customization improves predictability & quality-of-service**.
- **Example:**
  - Retrieving **accounting documents** (invoices, orders):
    - **Automatic retry** on failure.
    - **State remains unchanged** during retries.

- **What Are Complex Methods?**
  - **Extend basic HTTP methods** with additional rules
  - **Designed for specific interactions** where simple methods are insufficient
  - **Combine multiple functions into an aggregate interaction**

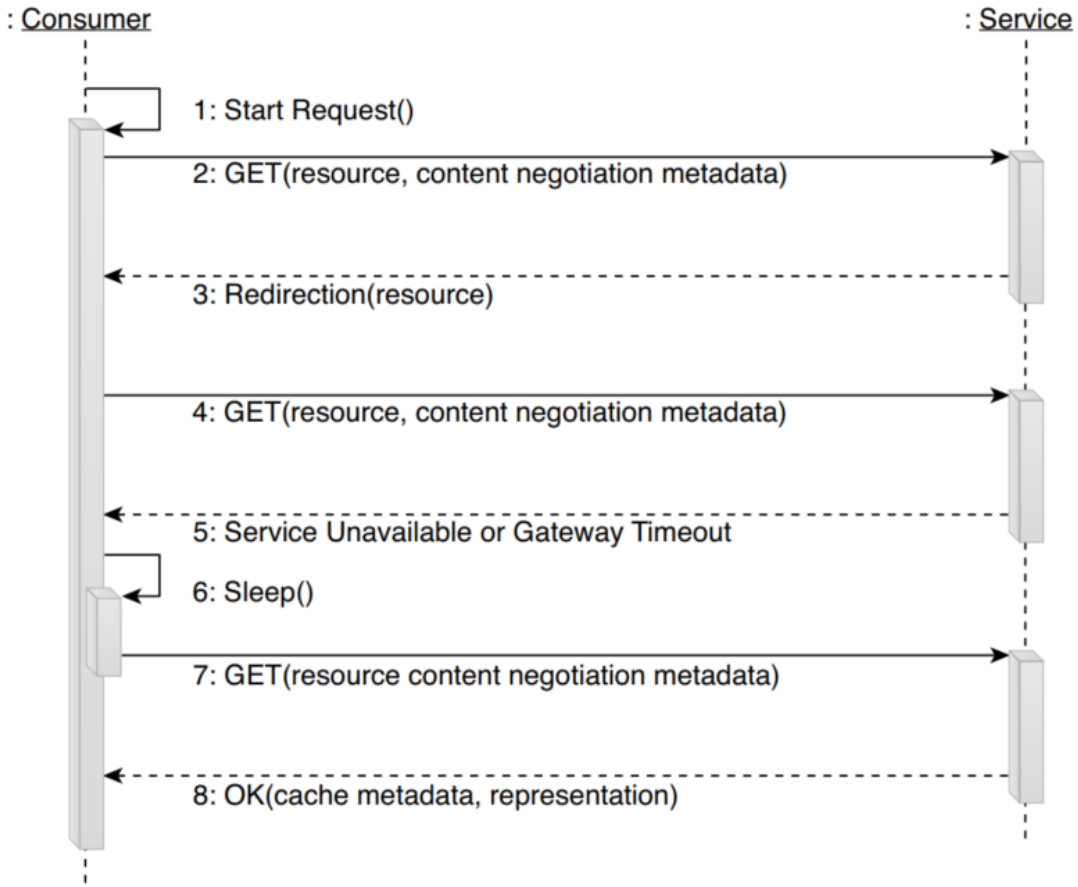
- **Benefit: Standardized system with custom logic for critical operations.**

Complex Method Design (2)

- **Complex methods encapsulate multiple primitive methods and additional logic**
- **Standardized within a service inventory for consistency**
- **Common complex methods**
  - **Fetch** – Recovers from errors during multiple GET requests
  - **Store** – Ensures reliable PUT or DELETE operations
  - **Delta** – Keeps consumers updated on changing resources
  - **Async** – Supports asynchronous request processing
- **Helps reduce redundant logic and simplifies maintenance**

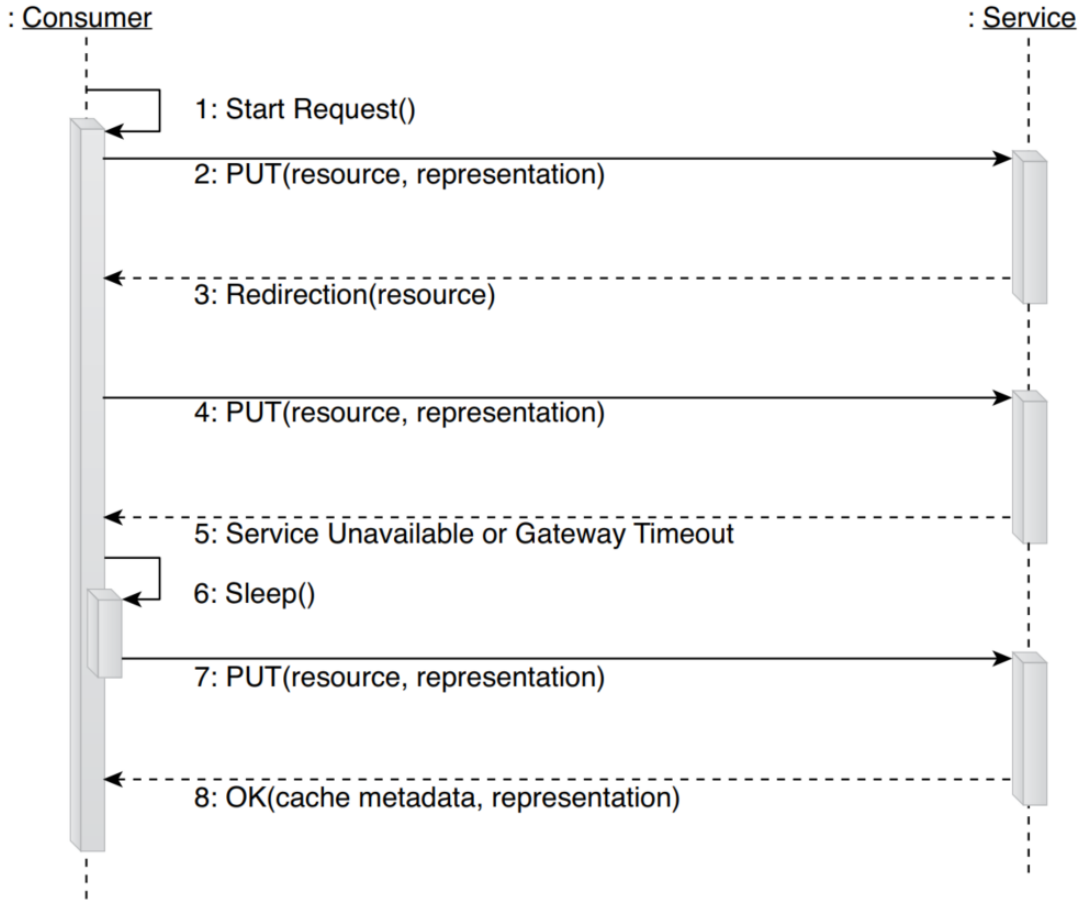
Stateless Complex Methods – Fetch Method

- *Stateless means there is no memory of the past*
- **Fetch Method enhances HTTP GET for reliable data retrieval**
  - **Automatic retries** on timeout or failure.
  - **Content negotiation** ensures compatibility
  - **Redirection handling** supports contract changes
  - **Cache control directives** optimize performance



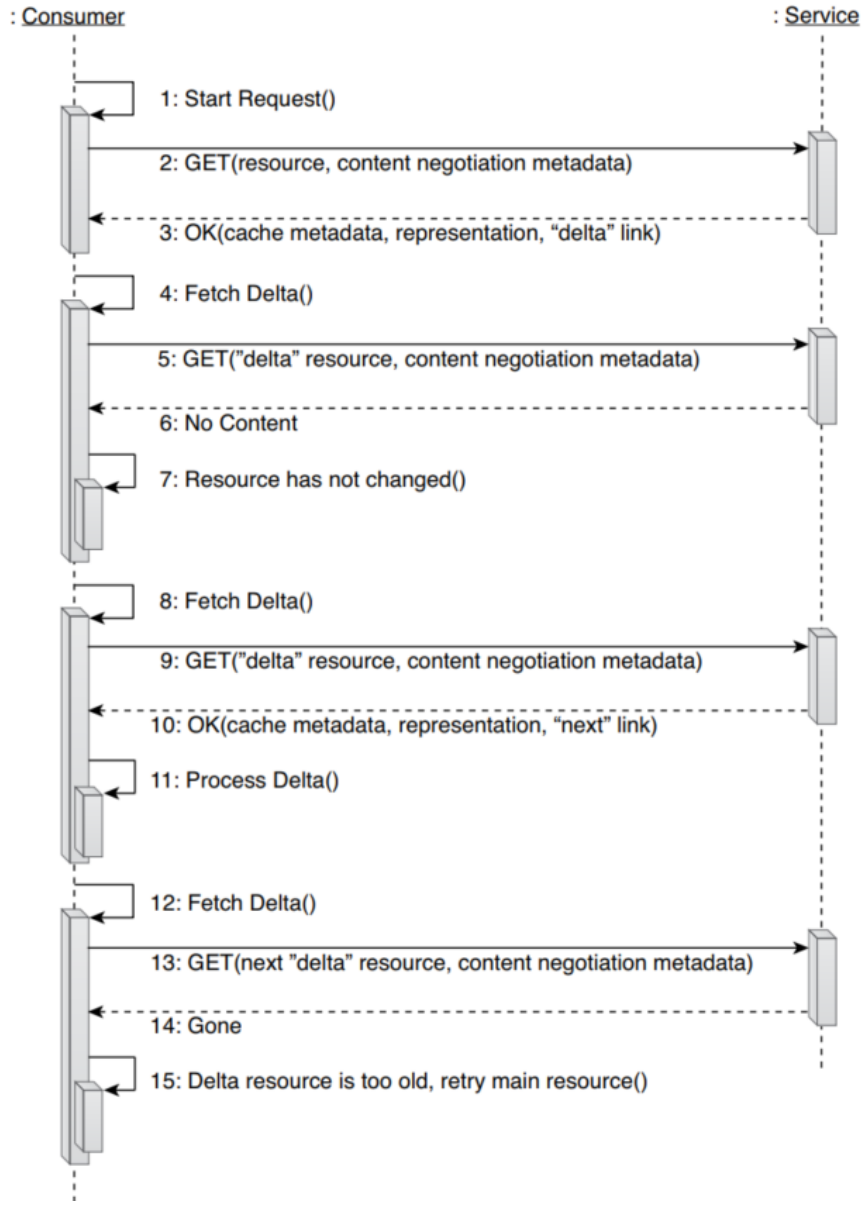
Stateless Complex Methods – Store Method

- **Store method enhances PUT and DELETE for resource updates.**
  - Provides **automatic retries, content negotiation, and redirection handling**
  - Ensures **idempotency**: repeating a successful request has no additional effect.
  - Helps handle **timeouts, low bandwidth, and failed requests** efficiently.



Stateless Complex Methods – Delta Method

- **Delta method keeps consumers synchronized with resource state changes.**
- **Mechanism**
  - **Service maintains a history** of changes.
  - Consumers retrieve **only the updates** since their last query.
  - Responses include **links to the next delta or indicate no changes**
- **Helps optimize bandwidth & reduce redundant data retrieval.**



Stateless Complex Methods – Async Method

- **Async method handles long-running requests without requiring an open connection.**
- **Mechanism**
  - The consumer sends a request **with a callback URL**.
  - The service responds with **202 Accepted** and may return a **tracking resource identifier**.
  - When processing completes, the service **sends the result to the callback URL**
  - The consumer can send a **DELETE request** to cancel an ongoing request

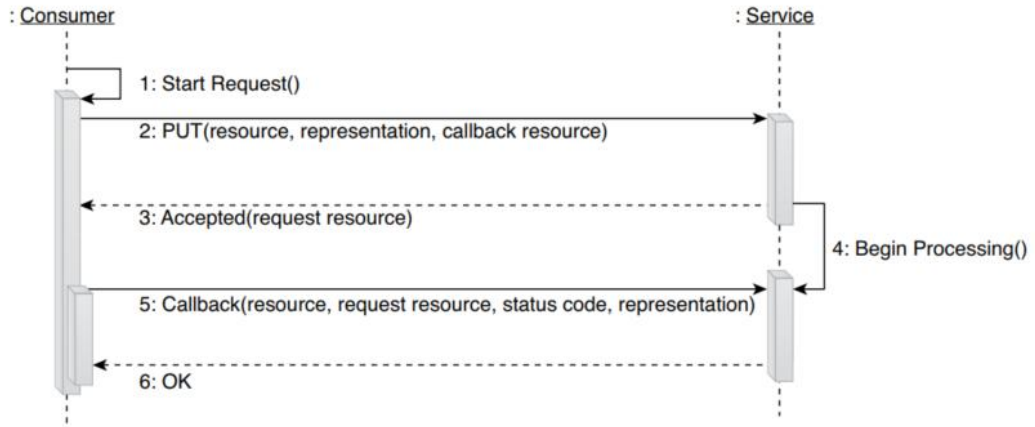


Figure 9.15
An asynchronous request interaction encompassed by the Async complex method.

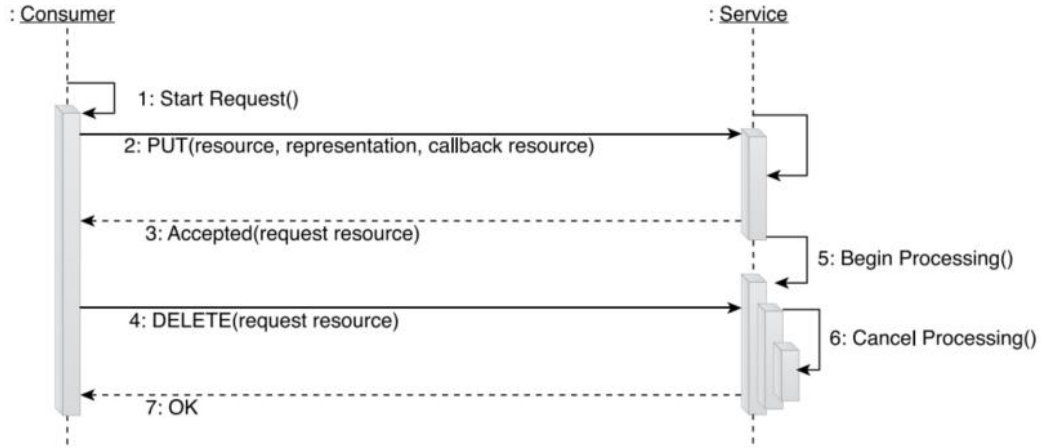


Figure 9.16
An asynchronous cancel interaction encompassed by the Async complex method.

Stateful Complex Methods – Trans Method

- **Trans method supports two-phase commit transactions between services**
  - Ensures **all changes succeed or all are rolled back.**
- **Introduces a custom PREP-PUT method (prepares changes without committing)**
- **Requires a transaction controller for commit/rollback management**

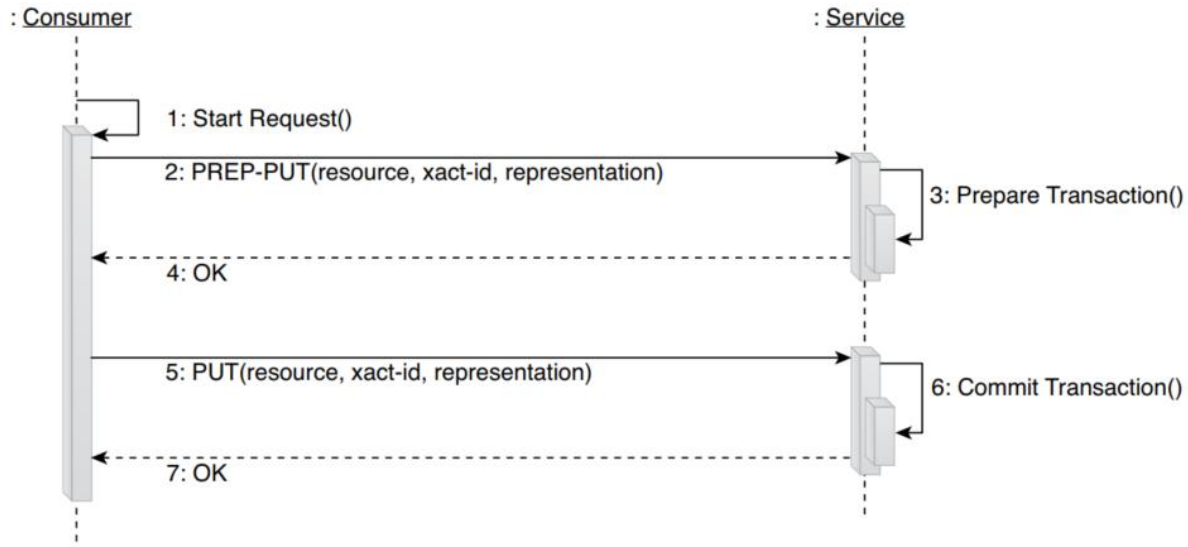
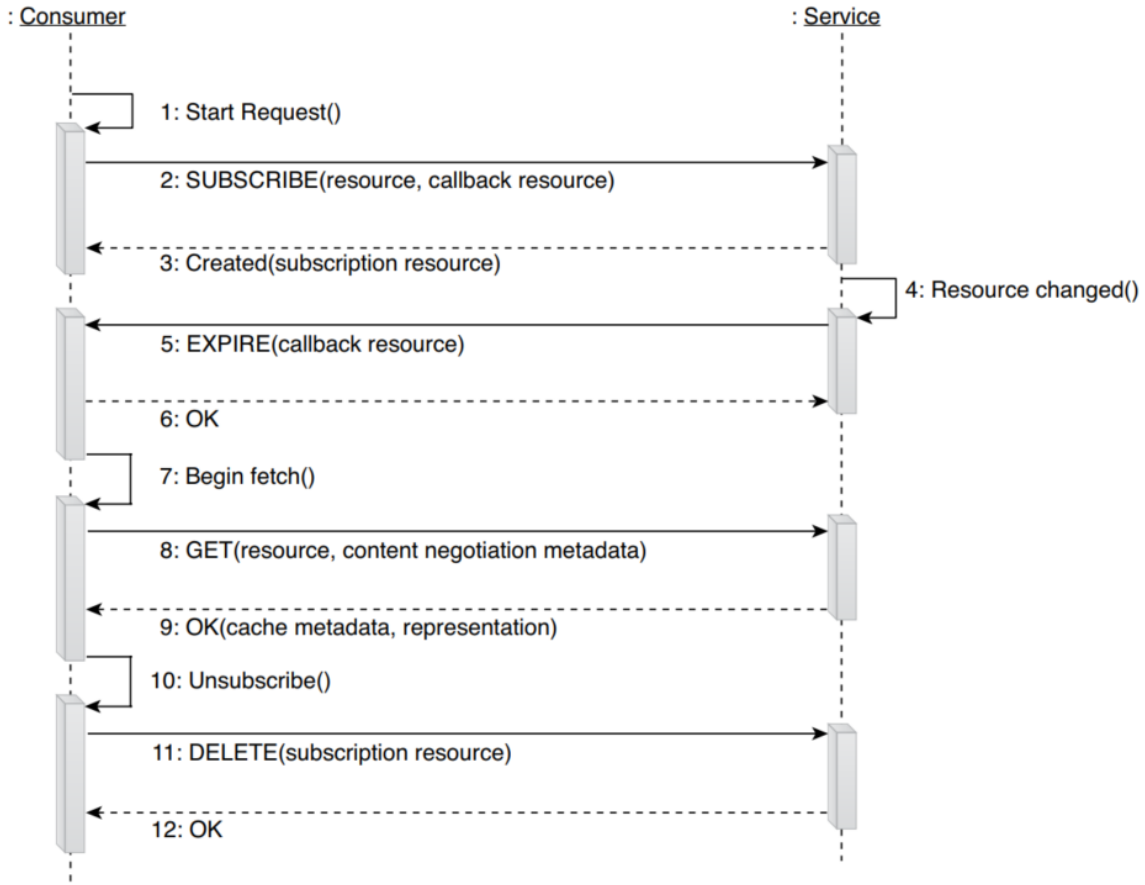


Figure 9.17
An example of a Trans complex method, using a custom primitive method called PREP-PUT.

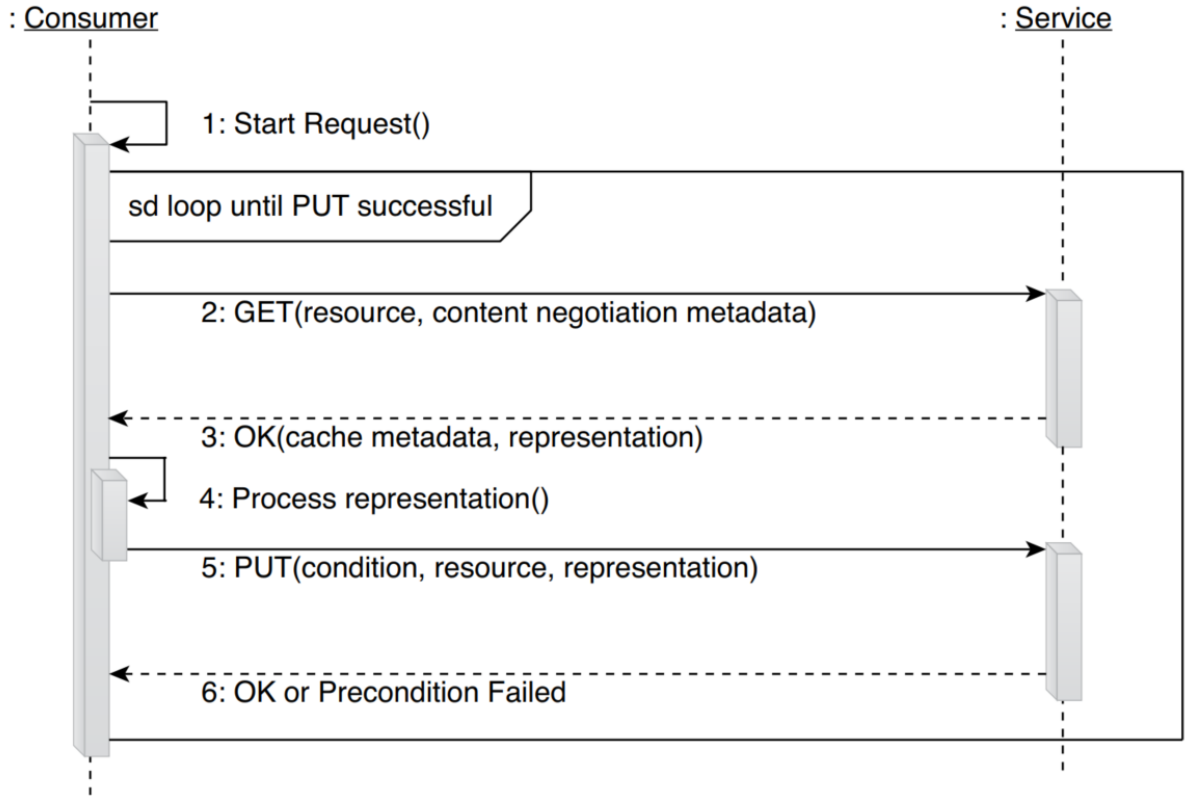
Stateful Complex Methods – PubSub Method

- **PubSub method supports real-time event-driven interactions**
- **Acts as a cache-invalidation mechanism**
  - Service **pushes notifications** about changes
  - Consumers **pull updates** using Fetch.
- **Minimizes state storage and reduces redundant fetches**
  - Can distribute (/topic) load **efficiently in cloud environments**



MUA Case study – OptLock Method

- **Challenge:** Concurrent updates to the same resource cause conflicts
- **Solution:** Introduce OptLock (Optimistic Locking) method.
- **Mechanism**
  - Consumer **fetches resource** → Receives **ETag** (version identifier).
  - Consumer **updates resource** → Sends **If-Match header** with ETag.
  - **If resource is unchanged**, update is accepted; **otherwise, rejected (409 Conflict)**.
- **Limitations:**
  - Works best for **low write-frequency resources**.
  - High concurrency can cause **frequent conflicts and retries**.

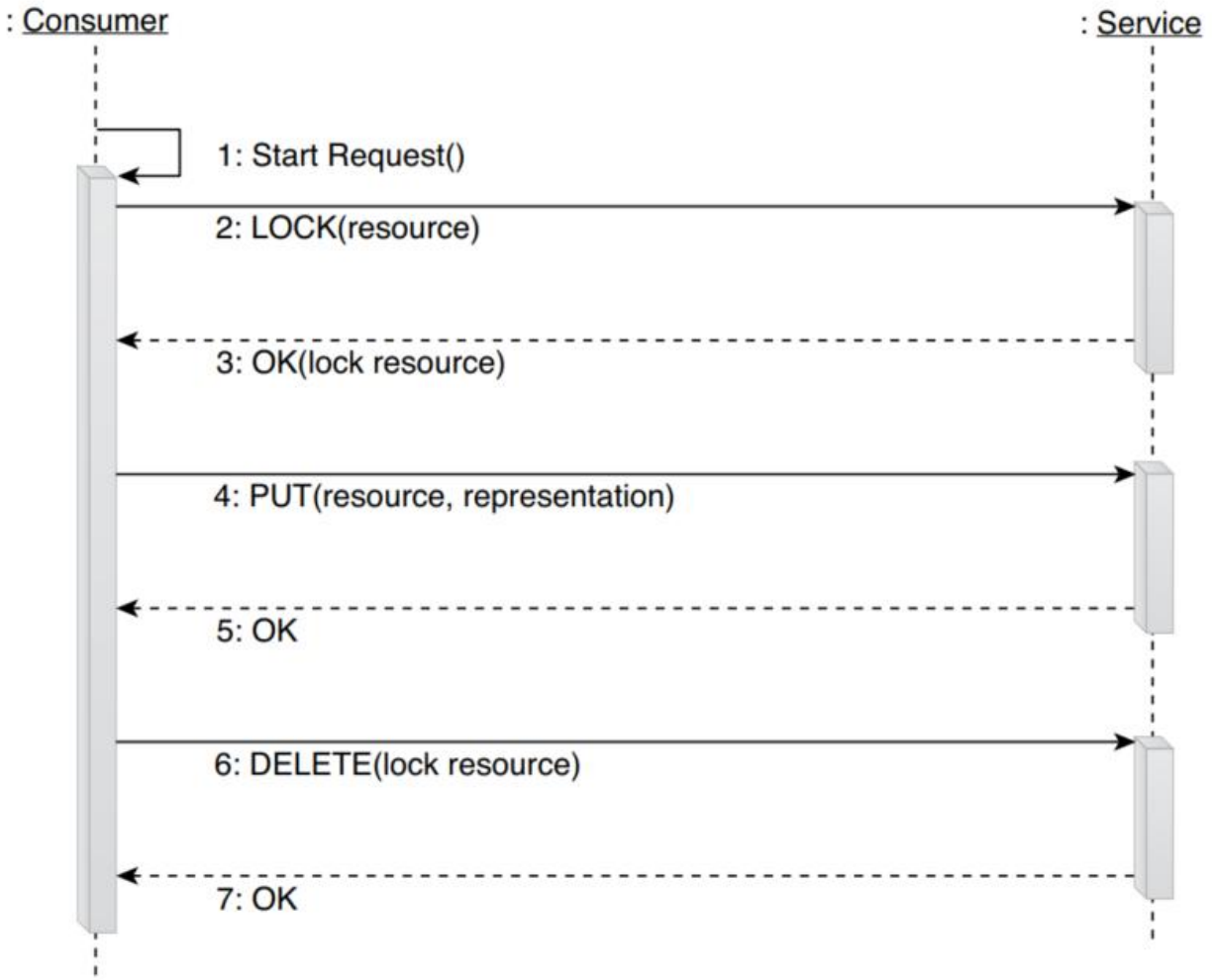


Need for Pessimistic Locking (PesLock Method)

- Problem with OptLock
  - Frequent **409 Conflict errors** during peak usage.
  - High concurrency makes **retries inefficient**
- **Proposed Solution**
  - Switch to **PesLock (Pessimistic Locking)**.
- PesLock ensures
  - **Exclusive access** to a resource during updates
  - Reduces **conflict risk** by preventing concurrent modifications.

MUA Case study – PesLock Method

- **PesLock ensures exclusive access but introduces stateful interactions**
  - Limits **concurrent access** while the lock is held
  - **Access control is required** to prevent unauthorized locks.
  - Locks must have **time out**
- **Considerations**
  - **Start with OptLock** for efficiency.
  - If **three retries fail**, fallback to **PesLock** for guaranteed execution.



Q & A